Broadview®
www.broadview.com.cn

深入理解 JVM & G1 GC

周明耀／著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

周明耀 ▶

- 12年投资银行项目、分布式计算项目工作经验，IBM开发者论坛专栏作者、InfoQ专栏作者。
- 一名IT技术狂热爱好者，一名顽强到底的工程师。推崇技术创新、思维创新，对于新技术非常热爱，致力于技术研发、研究，通过发布文章、书籍、互动活动的形式积极推广软件技术。
- 欢迎添加微信“michael_tec”，共同探讨IT技术话题。

深入理解 JVM & G1 GC

周明耀 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

G1 GC 提出了不确定性 Region，每个空闲 Region 不是为某个固定年代准备的，它是灵活的，需求驱动的，所以 G1 GC 代表了先进性。

本书主要为学习 Java 语言的学生、初级程序员提供 GC 的使用参考建议及经验，着重介绍了 G1 GC。中国的软件开发行业已经有几十年了，从目前的行业发展来看，单纯的软件公司很难有发展，目前流行的云计算、物联网企业实际上是综合性 IT 技术的整合，这就需要有综合能力的程序员。本书作者力求做到知识的综合传播，而不是仅仅针对 Java 虚拟机和 GC 调优进行讲解，也力求每一章节都有实际的案例支撑。本书具体包括以下几方面：JVM 基础知识、GC 基础知识、G1 GC 的深入介绍、G1 GC 调优建议、JDK 自带工具使用介绍等。

通读本书后，读者可以深入了解 G1 GC 性能调优的许多主题及相关的综合性知识。读者也可以把本书作为参考，对于感兴趣的主题，直接跳到相应章节寻找答案。

图书在版编目（CIP）数据

深入理解 JVM & G1 GC / 周明耀著. —北京：电子工业出版社，2017.6
ISBN 978-7-121-31468-1

Ⅰ. ①深… Ⅱ. ①周… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2017）第 097447 号

责任编辑：董　英
印　　刷：三河兴达印务有限公司
装　　订：三河兴达印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：15.5　　字数：337 千字
版　　次：2017 年 6 月第 1 版
印　　次：2017 年 6 月第 1 次印刷
印　　数：3000 册　　定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

目　录

序

这是我第一次为人写序，心中不免忐忑，引用余光中先生对于写序的感受：“我为人写序，于人为略而于文为详，用意也无非要就文本去探人本，亦即其艺术人格；自问与中国传统的序跋并不相悖，但手段毕竟不同了。”周明耀的书追求的是人本，我则尽力而为。

周明耀是我以前的同事，我们是认识十多年的朋友，同时我也是他婚礼的伴郎，见证了他的一步步成长。周明耀无论对工作、技术，或者社会生活，都有着自己独特的见解。他始终对技术充满了敬畏之心，学无止境，并付诸实践。因此，才有了这本书。

以我对 GC 的理解，想要学习 GC，首先需要理解为什么需要 GC。随着应用程序所应对的业务越来越庞大、复杂，用户越来越多，没有 GC 就不能保证应用程序的正常进行。而经常造成 STW 的 GC 又跟不上实际的需求，所以才会不断地尝试对 GC 进行优化。正如周明耀在文中描述的，HotSpot 有这么多的垃圾回收器（Serial GC、Parallel GC、Concurrent Mark Sweep GC），为什么还要发布 Garbage First（G1）GC？原因就是这个。

当今的商业模式，更多依赖市场的力量，不断淘汰旧的行业，把有限的资源让给那些竞争力更强、利润率更高的企业。类似地，硅谷也在不断淘汰过时的人员，从全世界吸收新鲜血液。经过半个多世纪的发展，在硅谷地区形成了只有卓越才能生存的文化理念。本着这样的理念，GC 承担了淘汰垃圾、保存优良资产的任务。正如周明耀所说，随着 G1 GC 的出现，GC 从传统的连续堆内存布局设计，逐渐走向不连续内存块，这是通过引入 Region 概念实现的，也就是说，由一堆不连续的 Region 组成了堆内存。其实也不能说是不连续的，只是它从传统的物理连续逐渐变为逻辑上的连续，这是通过 Region 的动态分配方式实现的，我们可以把一个 Region 分配给 Eden、Survivor、老年代、大对象区间、空闲区间中的任意一个，而不是固定它的作用，因为越是固定，越是呆板。

总的来说，本书对 Java GC 机制的分析深入浅出，是对大数据 Java 内存回收的优秀实践。读完茅塞顿开、受益匪浅。很多技术细节应用之后，对产品性能有明显提升。在此感谢周明耀的分享，希望他能够写出更多优秀的书籍。

华为南京研究所大数据产品部维护经理　吴骏

前　言

7岁那年，当我合上《上下五千年》一套三册全书时，我对自己说，我想当个作家。这一晃27年了，等待了27年，我的第一本书《大话Java性能优化》在2016年4月正式面世，2016年8月第二次印刷，感谢读者的厚爱。第一次印刷时出现一些错别字，请原谅编辑小姑娘，99万字对她来说确实太多了，这是我的责任，未来一定尽全力避免。《深入理解JVM & G1 GC》是我的第二本书，也即将面世。对于我的每一本书，我都怀着忐忑、惊喜的心情，就像第一次面对我的女儿“小顽子”，给她取这个小名，是希望她顽强到底，因为我相信，你若顽强到底，一切皆有可能。

我喜欢看书，每年购买的书接近100本，也喜欢技术积累。一直没有出书的想法，直到遇到了电子工业出版社的董老师，在深圳南湖的一席畅谈后，我决定做一位业余的技术作家，致力于中国软件开发行业的技术推广、普及、推动。

每年都要面试很多学生，我感觉中国的大学不太注重实际项目开发能力的培养，较为教条，这也是我的系统丛书首先从Java技术开始的原因，它更加接地气。本书主要为学习Java语言的学生、初级程序员提供JVM和GC的使用和优化建议及经验，力求做到知识的综合传播，而不是仅仅针对Java虚拟机调优进行讲解。本书具体包括以下几方面：JVM基础知识、GC基础知识、G1 GC的深入介绍、G1 GC调优建议、JDK自带工具使用介绍等。

本书基于JDK8，总的来说，没有一招鲜式的性能调优秘籍或包罗万象的性能百科，能让你摇身一变成为老练的GC性能调优专家。相当数量的GC性能问题还需要专门的知识技能才能解决。性能调优在很大程度上是一门艺术。解决的GC性能问题越多，技艺才会越精湛。我们不只要关心GC的持续演进，也要积极地去了解它的设计原理和设计目标。

最后，自我介绍一下，我叫周明耀，研究生学历，12年工作经验，IBM开发者论坛专家作

者。我是一名 IT 技术狂热爱好者，一名九三学社社员，一名顽强到底的工程师。我推崇技术创新、思维创新，对于新技术非常的热爱。

感谢我的家人，和谐的家庭帮助我完成了这本书。我的妻子，她美丽、细心、博学、偶尔不那么温柔，但是我很爱她。我的小顽子，她天生性格很像我，希望她能够踏踏实实做人，保持创新精神，平平安安、健健康康地生活下去。感谢我妻子的父母、我的父母，他们帮我照顾小孩，我才有时间编写此书。感谢浙江省特级教师、杭州高级化学老师郑克良老师，郑老师的一句永远不要放弃，推动着我多年的发展。感谢数学老师张老师在公开场合对我智商的褒奖，第一次收获这样的赞赏，对我这样内向的孩子是多么的重要，谢谢。

这本书献给我记忆中的爷爷奶奶、外公外婆，你们给我的都是最美的回忆。

我相信这本书不是终点，它是麦克叔叔此生一系列技术书籍的一员，咱们下一本书见。

1 第1章 JVM & GC 基础知识

单细胞生物在地球上经历了几十亿年的进化过程，又过了几十亿年，细胞与周围的细胞相互接触，从而形成了活的球形有机体，单细胞生物才进化为多细胞生物。在最初的时候，球形是多细胞生命能够生存的唯一形状，因为细胞之间只有相互接近才能互相协调功能。又过了十亿年，生命终于进化出了第一个神经元细胞，它是一种纤细的条状细胞，能够使两个细胞即使相隔一段距离仍能够通信。正是这项激活创新的诞生，各种各样的生命开始繁荣发展。有了神经元，生命不再局限于球状，细胞可以组成任何形状、大小和功能。蝴蝶、兰花和袋鼠，各种各样的生命形态都变成了可能。生命瞬间拓展出了成千上万种可能性，繁荣到令人惊叹，直到美丽的生命无处不在。程序设计的学习就好像探索生命起源一样，你不了解起源，就不可能找到正确的方法。[1]

JVM 是 Java 语言可以跨平台、保持高发展的根本，没有了 JVM，Java 语言将失去运行环

[1] 推荐带孩子去看《冰川时代》这本系列动画电影，它不只是动画片，也是对于大自然的思考，中国动画片倾向于培养人的三观，美国人的动画片更倾向于让人解释自己的行为，自己约束自己的行为。

境。针对 Java 程序的性能优化一定不可能避免针对 JVM 的调优，随着 JVM 的不断发展，我们的应对措施也在不断地跟随、变化，内存的使用逐渐变得越来越复杂。所有高级语言都需要垃圾回收机制的保护，所以 GC 就是这么重要。

本章主要介绍和解决以下问题，这些也是全书的基础。

- 为什么我们需要了解 JVM 和 GC，这是您阅读本书的依据。
- 了解 GC 的基础常用术语知识，作者和读者需要对术语定义进行统一。
- 了解 JVM 的基础知识，包括堆、栈、方法区等。
- 为深入了解 JVM 和 GC 做好知识储备。

1.1 引言

还记得机器猫吗？他和康夫有一张书桌，书桌的抽屉其实是一个时空穿梭通道，让我们操作机器猫[1]的时空机器，回到 1998 年。那年的 12 月 8 日，第二代 Java 平台的企业版 J2EE[2]正式对外发布。为了配合企业级应用落地，1999 年 4 月 27 日，Java 程序的舞台——Java HotSpot Virtual Machine（以下简称 HotSpot[3]）正式对外发布，并从这之后发布的 JDK1.3 版本开始，HotSpot 成为 Sun JDK 的默认虚拟机。

既然有了虚拟机，那一定需要收集垃圾的机制，这就是 Garbage Collection[4]，对应的产品我们称为 Garbage Collector，垃圾回收器。1999 年伴随 JDK1.3.1 一起来的是串行方式的 Serial GC[5]，它是第一款 GC，并且这只是起点。此后，JDK1.4 和 J2SE1.3 相继发布。2002 年 2 月 26 日，J2SE1.4 发布，Parallel GC[6]和 Concurrent Mark Sweep[7]（CMS）GC 跟随 JDK1.4.2 一起发布，并且 Parallel GC 在 JDK6 之后成为 HotSpot 默认 GC。

---

[1] 由日本漫画家藤本弘（笔名藤子・F・不二雄）和安孙子素雄（笔名藤子不二雄 A）共同创作的漫画作品。笔者最喜欢的动画片。

[2] 全称 Java 2 Platform Enterprise Edition，是一套全然不同于传统应用开发的技术架构，包含许多组件，主要可简化且规范应用系统的开发与部署，进而提高可移植性、安全与再用价值。

[3] 来源于无线热点。原指在公共场所提供无线局域网（Wi Fi）接入 Internet 服务的地点。

[4] 其实每种高级语言都有这个机制，不单单是针对 Java 语言的，推荐大家阅读 Richard Jones 等人编著的 *The Garbage Collection Handbook* 这本书，最好读英文原版的。

[5] 一种单线程的 GC。

[6] 与 Serial GC 一样都基于分代概念，差别是它采用了多线程方式加速运行垃圾回收。

[7] 是一款让应用程序可以和 GC 分享处理器资源的设计落地产品。

HotSpot 有这么多的垃圾回收器，那么如果有人问，Serial GC、Parallel GC、Concurrent Mark Sweep GC 这三个 GC 有什么不同呢？

- 如果你想要最小化地使用内存和并行开销，请选 Serial GC。
- 如果你想要最大化应用程序的吞吐量，请选 Parallel GC。
- 如果你想要最小化 GC 的中断或停顿时间，请选 CMS GC。

那么问题来了，既然我们已经有了上面三个强大的 GC，为什么还要发布 Garbage First（G1）GC 呢？我只能说：“人类的追求是无限的，就像女人永远追求更美丽。”

为什么名字叫作 Garbage First（G1）呢？故事背景稍微讲解一下，具体的内容请读者看后续章节。因为 G1 是一个并行回收器，它把堆内存分割为很多不相关的区间（Region），每个区间可以属于老年代或者年轻代，并且每个年龄代区间可以是物理上不连续的。老年代区间这个设计理念本身是为了服务于并行后台线程，这些线程的主要工作是寻找未被引用的对象[1]，而这样就会产生一种现象，即某些区间的垃圾（未被引用对象）多于其他的区间。我们后面会介绍，垃圾回收时都是需要停下应用程序的，不然就没有办法防止应用程序的干扰[2]，然后 G1 GC 可以集中精力在垃圾最多的区间上，并且只费一点点时间就可以清空这些区间里的垃圾，腾出完全空闲的区间。绕来绕去终于明白了，由于这种方式的侧重点在于处理垃圾最多的区间，所以我们给 G1 一个名字：垃圾优先（Garbage First）。

G1 内部主要有四个操作阶段，这四个阶段也是第 4 章和第 5 章的主要内容，即：

- 年轻代回收（A Young Collection）；
- 运行在后台的并行循环（A Background，Concurrent Cycle）；
- 混合回收（A Mixed Collection）；
- 全量回收（A Full GC）。

1.2　基本术语

结合我上一本书出版后的反馈信息以及个人阅读专业书籍的经验，由于计算机程序语言全部都是由国外创造、开发的，所以很多单词是经中国的技术专家由英文单词翻译的，这样可能的结果是专业术语按照自己的理解翻译，版本会较多，就如同各地方言一样。我觉得本书的专

[1] 在 GC 看来，未被引用对象是可以被回收的。

[2] 因为应用程序会不断地生产垃圾。警察办案时不是也要清场嘛，原因是一致的。

业性较强，很多英文单词容易出现歧义，所以我觉得有必要在这里统一一下对专业术语的中英文对照，以及对应的解释。我翻译和理解得不一定很到位，所以事先统一中英文对照还是很有必要的。

另外，需要提前说明的是，为了让每一个章节相对独立，有利于读者跳过一些章节，所以一些知识可能会重复介绍，这不是失误，是为了实现我对于我所有出版读物的规划，让技术以实践为目标，让使用更简单、更易落地。

1.2.1 Java 相关术语

1.2.1.1 Millisecond

GC 内部的动作（某一个过程）一般都是在毫秒级完成。毫秒（ms）是一种较为微小的时间单位，是一秒的千分之一。除了毫秒以外，还有皮秒、纳秒、微秒，计算公式分别如下：

- 皮秒，符号 ps（英语：picosecond，1 皮秒等于一万亿分之一秒（10^{-12} 秒）

1,000 皮秒 = 1 纳秒；1,000,000 皮秒 = 1 微秒；1,000,000,000 皮秒 = 1 毫秒；1,000,000,000,000 皮秒 = 1 秒。

- 纳秒，符号 ns（英语：nanosecond ）

1 纳秒等于十亿分之一秒（10^{-9} 秒）；1 纳秒 = 1000 皮秒；1,000 纳秒 = 1 微秒；1,000,000 纳秒 = 1 毫秒；1,000,000,000 纳秒 = 1 秒。

- 微秒，符号 μs（英语：microsecond ）

1 微秒等于一百万分之一秒（10^{-6} 秒）；0.000 001 微秒 = 1 皮秒；0.001 微秒 = 1 纳秒；1,000 微秒 = 1 毫秒；1,000,000 微秒 = 1 秒。

- 毫秒，符号 ms（英语：millisecond ）

1 毫秒等于一千分之一秒（10^{-3} 秒）；0.000 000 001 毫秒 = 1 皮秒；0.000 001 毫秒 = 1 纳秒；0.001 毫秒 = 1 微秒；1000 毫秒 = 1 秒。

1.2.1.2 Megabyte

GC 的区间划分基数一般采用兆级别，所以这里需要解释 MB。

MB（全称 MByte）：计算机中的一种储存单位，读作“兆”。数据单位 MB 与 Mb（注意 B 字母的大小写）常被误认为是一个意思，其实 MByte 含义是“兆字节”，Mbit 的含义是“兆

比特”。MByte 是指字节数量，Mbit 是指比特位数。MByte 中的“Byte”虽然与 Mbit 中的“bit”翻译一样，都是比特，也都是数据量度单位，但二者是完全不同的。Byte 是“字节数”，bit 是“位数”，在计算机中每八位为一字节，也就是 1Byte=8bit，是 1∶8 的对应关系。因此在书写单位时一定要注意 B 字母的大小写和含义。

1.2.1.3　Java 应用程序如何配置 JVM 参数

这个实践方式需要贯穿整本书，毕竟只有学会配置参数（JVM 选项），才能查看 GC 日志，最终才能主导讨论 GC 优化方法。

这里不是说如何配置正确的参数，仅仅只是演示一下如何在 Eclipse[1]和 Linux[2]这两个普遍运行 Java 程序的场景如何配置 JVM 参数（后面都统一称为选项）。

Eclipse 工具的使用方法如下：

选择 Eclipse → Run → Run Configurations → Arguments → VM arguments，在左边 Java Application 栏选中要设置的 Project 运行类，在 VM arguments 中填入 -Xms64m -Xmx256m。

这里-Xms 是设置内存初始化的大小（如上面的 64m），而-Xmx 是设置最大能够使用内存的大小（如上面的 256m，最好不要超过物理内存）。配置方式如图 1-1 所示，演示代码如代码清单 1-1 所示，运行输出请见代码清单 1-2 和代码清单 1-3。

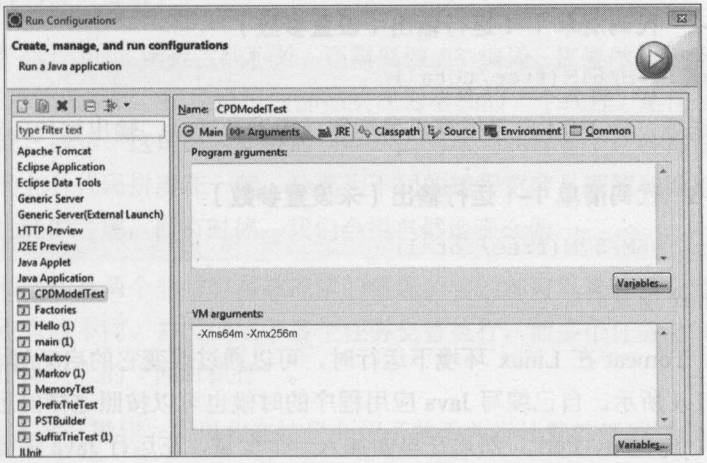


图 1-1　Eclipse 配置参数演示图

[1] 一种开源的 IDE 工具，原为 IBM 公司内部产品，下载地址 https://eclipse.org/downloads/。

[2] Linux 有一个蔚为壮观的生态系统，包括形形色色的贡献者、提供商和用户，这里泛指操作系统。

代码清单 1-1　演示代码

```
public class JVMDemoTest {
    /**
     * 获取当前 jvm 的内存信息
     * @return
     */
    public static String toMemoryInfo() {

      Runtime runtime = Runtime.getRuntime ();
      int freeMemory = ( int ) (runtime.freeMemory() / 1024 / 1024);
      int totalMemory = ( int ) (runtime.totalMemory() / 1024 / 1024);
      return freeMemory + "M/" + totalMemory + "M(free/total)" ;
    }
    /**
     *
     * @param args
     */
    public static void main(String[] args) {
      System.out.println( "memory info :" + toMemoryInfo ());
    }
}
```

代码清单 1-2　代码清单 1-1 运行输出（设置参数）

```
memory info :60M/61M(free/total)
```

如果不配置选项，让 JVM 自己根据机器配置使用默认选项，输出如代码清单 1-3 所示。

代码清单 1-3　代码清单 1-1 运行输出（未设置参数）

```
memory info :56M/57M(free/total)
```

下面介绍 Linux 场景。

举一个例子，Tomcat 在 Linux 环境下运行时，可以通过改变它的启动脚本来添加 JVM 参数，如代码清单 1-4 所示。自己编写 Java 应用程序的时候也可以按照这样的思路，编写一个脚本，可以是 Shell、Python、Perl[1]，然后在里面加入一个变量，在运行 Java 关键字的时候将这个关键字加为运行属性。

[1] Linux 的脚本三兄弟。

代码清单 1-4　Linux 配置参数示例

```
在/usr/local/apache-tomcat-5.5.23/bin 目录下的 catalina.sh
添加:JAVA_OPTS='-Xms512m -Xmx1024m'
或者 JAVA_OPTS="-server -Xms800m -Xmx800m
-XX:MaxNewSize=256m" 或者 CATALINA_OPTS="-server -Xms256m -Xmx300m"
```

再次声明，以后都用选项代替参数。

1.2.1.4　并行计算

服务端程序与一般的用户终端程序相比，服务端程序需要承受很重的用户访问压力。根据淘宝的数据，“双 11”一天支付宝核心数据库集群处理了 41 亿个事务，执行了 285 亿次 SQL，生成了 15TB 日志，访问了 1931 亿次内存数据块、13 亿个物理读。如此密集的访问，恐怕任何一台单机都难以胜任，因此，并行程序也就自然成了一个出路。

面对复杂业务模型，并行程序会比串行程序更容易适应业务需求，更容易模拟我们的现实世界。毕竟，我们的世界本质上是并行的。比如，当你开开心心去上学的时候，妈妈可能在家里忙着家务，爸爸在外打工赚钱，一家人其乐融融。如果有一天，你需要使用你的计算机来模拟这个场景，你会怎么做呢？如果你在一个线程里，既做了你自己，又做了妈妈，又做了爸爸，显然这不是一种好的解决方案。但如果你使用三个线程，分别模拟这三个人，一切看起来又是那么自然，而且容易被人理解。

虚拟机除了要执行 main 函数主线程外，还需要做 JIT 编译，需要做垃圾回收。无论是 main 函数、JIT 编译还是垃圾回收，在虚拟机内部都实现为单独的一个线程。是什么使得虚拟机的研发人员这么设计呢？显然，这是因为业务的需要。因为这里的每一个任务都是相对独立的，不应该将没有关联的业务代码拼凑在一起，分离为不同的线程更容易理解和维护。因此，使用并行也不完全出自性能的考虑，而有时候，我们会很自然地那么做。

注意，并发和并行是两个非常容易被混淆的概念。它们都可以表示两个或者多个任务一起执行，但是偏重点有些不同。并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

GC 里面有很多并行操作，所以我在这里介绍了关于并行计算的基础概念。

1.2.1.5　进程和线程

计算机内部每个正在系统上运行的程序都是一个进程，每个进程包含一到多个线程。进程也可能是整个程序或者是部分程序的动态执行。线程是一组指令的集合，或者是程序的特殊段，

它可以在程序里独立执行，也可以把它理解为代码运行的上下文。所以线程基本上是轻量级的进程，它负责在单个程序里执行多任务。通常由操作系统负责多个线程的调度和执行。

线程是程序中一个单一的顺序控制流程，在单个程序中同时运行多个线程完成不同的工作称为多线程。

线程和进程的区别在于，子进程和父进程有不同的代码和数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文。多线程主要是为了节约CPU时间，对于如何发挥利用，需要根据具体情况而定。注意，线程在运行中需要使用计算机的内存资源和CPU。

1.2.1.6 多线程（multithreading）

这个知识点也会贯穿整个GC知识点，毕竟多线程是所有高级语言都支持的特性。

经常会听到公司的HR说，“我们不需要纯管理人员”。我以前也有一段时间是这样认为的，后来我发现，其实我们是需要管理人员的。我们需要那种具备技术能力，理解技术、人文、产品、沟通，具有多任务同时调度能力的管理人员[1]。也就是说，还是需要管理人员的，否则就无法有效、高效地调动研发团队，研发团队的目标是既能够做出很好的产品、符合市场的要求，也能够快速解决市场反馈的缺陷。这和多线程管理方式类同。

多线程，这是一种从软件或者硬件上实现多个线程并发执行的技术，一般高级语言，特别是面向对象语言都具有该特性。具有多线程能力的计算机因为有硬件支持而能够在同一时间执行多于一个线程，从而提升整体处理的性能。具有这种能力的处理器包括对称多处理机、多核心处理器以及芯片级多处理（Chip-level Multithreading）或同时多线程（Simultaneous Multithreading）处理器。

如图1-2所示，描绘了一个Java线程的生命周期，从出生→开发→运行→死亡，中间还存在一些临时性状态，例如等待。

[1] 笔者认为乔布斯就是那样的人，他理解技术，只是从不编程。

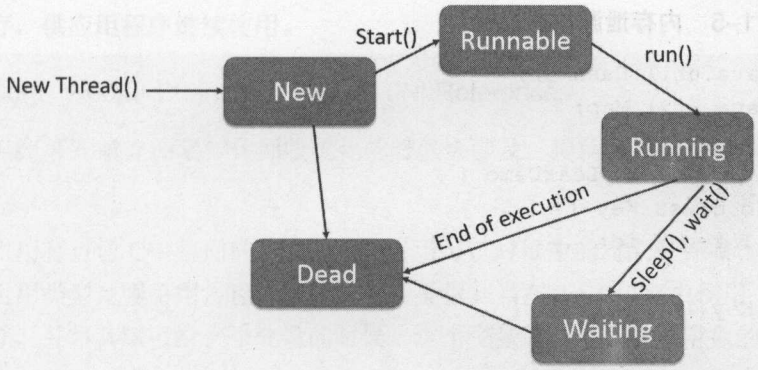


图 1-2　Java 线程生命周期图

1.2.1.7　内存泄漏（Memory Leak）

这个概念会在后续 GC 抛出异常时需要介绍，较为常见的 OutOfMemory 异常是最基本的内存不足引起的异常，很多时候是因为应用程序造成的内存泄漏情况引起了该错误的发生。

有句话叫作“管理浮于表面”，指的是没有有效的管理措施，即便制定了，也没有真正落实，这样就容易出现人浮于事的情况，就会有很多烂摊子没有人去收拾。总的来说，抓细节永远是最难的，这和解决内存泄漏有得比。

内存泄漏也称作“存储渗漏”，用动态存储分配函数动态开辟的空间，在使用完毕后未被释放，结果导致一直占据该内存单元，一直持续到程序结束，即所谓内存泄漏[1]。

内存泄漏形象的比喻是“操作系统可提供给所有进程的存储空间正在被某个进程榨干”，最终结果是程序运行时间越长，占用存储空间越多，最终用尽全部内存空间，造成整个系统崩溃。所以“内存泄漏”是从操作系统的角度来看的。这里的存储空间并不是指物理内存，而是指虚拟内存大小，这个虚拟内存大小取决于磁盘交换区设定的大小。由程序申请的一块内存，如果没有任何一个指针指向它，那么这块内存就泄漏了。

在 Java 程序里，如果发生内存泄漏，那么最后都会抛出 OutOfMemoryError 异常，如代码清单 1-5 所示，模拟了一个抛出 OOM 异常的程序。

[1] 其实说白了就是该内存空间使用完毕之后未回收。

代码清单 1-5　内存泄漏演示代码

```
import Java.util.HashMap;
import Java.util.Map;

public class MemoryLeakDemo {
    static class Key {
         Integer id;

         Key(Integer id) {
            this.id = id;
         }

         @Override
         public int hashCode() {
            return id.hashCode();
         }
       }

       public static void main(String[] args) {
          Map m = new HashMap();
          while (true)
             for (int i = 0; i < 10000; i++)
                if (!m.containsKey(new Key(i)))
                   m.put(new Key(i), "Number:" + i);
       }
}
```

运行后会看到如代码清单 1-6 所示的错误日志输出。

代码清单 1-6　内存泄漏演示代码运行输出

```
   Exception in thread "main" Java.lang.OutOfMemoryError: GC overhead limit
exceeded
       at Java.util.HashMap.newNode(Unknown Source)
      at Java.util.HashMap.putVal(Unknown Source)
      at Java.util.HashMap.put(Unknown Source)
      at MemoryLeakDemo.main(MemoryLeakDemo.Java:23)
```

由于 GC 一直在发展，所以一般情况下，除非应用程序占用的内存增长速度非常快，造成垃圾回收已经跟不上内存消耗的速度，否则不太容易出现 OOM 的情况。大多数情况下，GC 会进行各种年龄段的垃圾回收，实在不行了就放大招，来一次独占式的 Full GC 操作，这时候会

回收大量的内存，供应用程序继续使用。

1.2.1.8　Soft、Weak、Phantom、Final 和 JNI References

这个知识点在第 6 章介绍老年代回收优化的时候会涉及，提前介绍引用类型，对于整本书也有一些意义。

Java 中的引用有点像 C++里面的指针，通过引用可以对堆中的对象进行操作。在 Java 程序中，最常见的引用类型是强引用，也是默认的引用类型。当在 Java 语言中使用 New 操作符创建一个新的对象，并将其赋值给一个变量的时候，这个变量就成为指向该对象的一个强引用。

在前面提到过，判断一个对象是否存活的标准为是否存在指向这个对象的引用[1]。在某函数中创建对象，例如：StringBuffer str = new StringBuffer(“Hello World”);，假设该代码是在函数体内运行的，那么局部变量 str 将被分配在栈内，而对象 StringBuffer 实例，被分配在 Java 堆上。局部变量 str 指向 StringBuffer 实例所在堆空间，通过 str 可以操作该实例，那么 str 就是 StringBuffer 的引用。此时，如果运行一个赋值语句：StringBuffer str1=str;，那么，str 所指向的对象也将被 str1 所指向，同时在局部栈空间上会分配空间存放 str1 变量。此时，StringBuffer 实例就有两个引用，而对引用使用“==”操作用于表示两个操作数所指向的堆空间地址是否相同，不表示两个操作数所指向的对象是否相等。

Java 中提供了 4 个级别的引用，即强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）这 4 个级别。在这 4 个级别中只有强引用类是包内可见的，其他 3 种引用类型均为 Public，可以在应用程序中直接使用，垃圾回收器会尝试回收只有弱引用的对象。

简要罗列一个 GC 对于不同引用类型的区别。

- **强引用（Strong Reference）**：在一个线程内，无需引用直接可以使用的对象，除非引用不存在了，否则强引用不会被 GC 清理。我们平时声明变量使用的就是强引用，普通系统 99%以上都是强引用，比如，String s = "Hello World"。
- **软引用（Soft Reference）**：JVM 抛出 OOM 之前，GC 清理所有的软引用对象。垃圾回收器在某个时刻决定回收软可达的对象的时候，会清理软引用，并可选地把引用存放到一个引用队列（Reference Queue）[2]。类似弱引用，只不过 Java 虚拟机会尽量让软引

[1] GC 内部判断是否回收一个对象，基本也是使用这条规律。

[2] 具有可以在弱引用的对象被垃圾收集时发出通知的作用。

用的存活时间长一些，迫不得已才清理。

- **弱引用（Weak Reference）**：弱引用对象与软引用对象的最大不同就在于，当 GC 在进行回收时，需要通过算法检查是否回收软引用对象，而对于弱引用对象，GC 总是进行回收。弱引用对象更容易、更快被 GC 回收。虽然，GC 在运行时一定回收弱引用对象，但是复杂关系的弱对象群常常需要好几次 GC 的运行才能完成。就像上面描述的场景，弱引用对象常常用于 Map 结构中，引用数据量较大的对象，一旦该对象的强引用为 null 时，GC 能够快速地回收该对象空间。
- **虚引用（Phantom Reference）**：又称为幽灵引用，主要目的是在一个对象所占的内存被实际回收之前得到通知，从而可以进行一些相关的清理工作。幽灵引用在使用方式上与之前介绍的三种引用类型有很大的不同。首先幽灵引用在创建时必须提供一个引用队列作为参数，其次幽灵引用对象的 get 方法总是返回 null，因此无法通过幽灵引用来获取被引用的对象。

1.2.1.9 finalization 机制

Java 语言提供了对象终止（finalization）机制来允许开发人员提供对象被销毁之前的自定义处理逻辑。Object 类提供了 finalize 方法来添加自定义的销毁逻辑。如果一个类有特殊的销毁逻辑，可以覆写 finalize 方法。

从功能上来说，finalize 方法与 C++中的析构函数比较相似，但是 Java 采用的是基于垃圾回收器的自动内存管理机制，所以 finalize 方法在本质上不同于 C++中的析构函数。当垃圾回收器发现没有引用指向一个对象时，会调用这个对象的 finalize 方法。通常在这个方法中进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。

由于 finalize 方法的存在，虚拟机中的对象一般处于三种可能的状态。第一种是可达状态，当有引用指向该对象时，该对象处于可达状态。根据引用类型的不同，有可能处于强引用可达、软引用可达或弱引用可达状态。第二种是可复活状态，如果对象的类覆写了 finalize 方法，则对象有可能处于该状态。虽然垃圾回收器是在对象没有引用的情况下才调用其 finalize 方法，但是在 finalize 方法的实现中可能为当前对象添加新的引用。因此在 finalize 方法运行完成之后，垃圾回收器需要重新检查该对象的引用。如果发现新的引用，那么对象会回到可达状态，相当于该对象被复活，否则对象会变成不可达状态。当对象从可复活状态变为可达状态之后，对象会再次出现没有引用存在的情况。在这个情况下，finalize 方法不会被再次调用，对象会直接变成不可达状态，也就是说，一个对象的 finalize 方法只会被调用一次。第三种是不可达状态，在这个状态下，垃圾回收器可以自由地释放对象所占用的内存空间。

1.2.1.10　Serviceability Agent（SA）

这个工具在第 6 章会具体讲解其基本原理，也会重点介绍 SA 工具的使用方法及优缺点。

SA（Serviceability Agent）是 JDK 自带的不为广大 Java 程序员所熟悉的底层诊断工具[1]。酒香不怕巷子深，SA 提供了一套可以深入 JVM 内部进行探索的机制，对于 Java Web 应用、Java 服务端应用的各种问题的诊断具有重要意义。

平时用来查看 VM 内部信息的常用的工具都在$JAVA_HOME/bin 目录下，如图 1-3 所示，其中一些工具就是用 Serviceability Agent[2]开发的。

SA 与目标进程是两个独立的进程，这个必须明确，所以两个进程之间通过进程间通信实现调试，并且 SA 不会影响目标进程的正常运行。SA 依赖于操作系统提供的调试 API，属于建立在一系列的 Debug 原语上的工具。

这里不多介绍，我们会在第 6 章详细介绍使用方案、方法、输出。

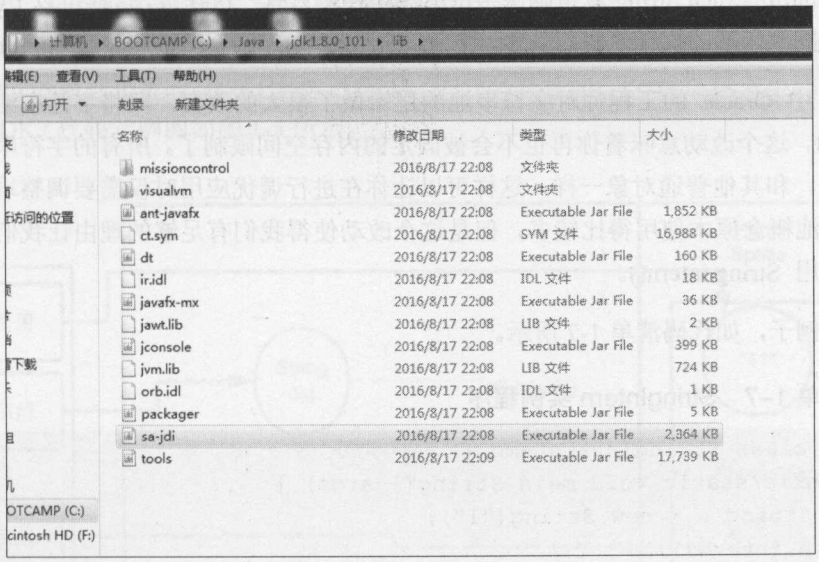


图 1-3　SA jar 包位置图

---

[1] 国内程序员工作压力较大，以完成任务为第一目标，有些人没有太多时间深入研究技术。

[2] Oracle 官网解释：The Serviceablity Agent is collection of Sun internal code that aids in debugging HotSpot problems. It is also used by several JDK tools - jstack, jmap, jinfo, and jdb. See SA for more information。即一个内部调试工具的集合。官方 API 地址：https://docs.oracle.com/javase/8/docs/serviceabilityagent/。

1.2.1.11 Interned Strings[1]

这个专业术语会在第 3 章讲解选项（参数）-XX:+UseStringDeplucation 时提到，G1 GC 专门针对这个问题提出了一个 JVM 选项。

在 Java 语言中有 8 种基本类型和一种比较特殊的类型 String。这些类型为了使它们在运行过程中速度更快、更节省内存，都提供了一种常量池的概念。常量池就类似一个 Java 系统级别提供的缓存。8 种基本类型的常量池都是系统协调的，String 类型的常量池比较特殊。它的主要使用方法有两种。

- 直接使用双引号声明出来的 String 对象会直接存储在常量池中。
- 如果不是用双引号声明的 String 对象，可以使用 String 提供的 intern 方法。intern 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中。

通俗点讲，Interned String 就是确保字符串在内存里只有一份拷贝，这样可以节约内存空间，加快字符串操作任务的执行速度。注意，这个值会被存放在字符串内部池（String Intern Pool）。

Java 7 中 Oracle 的工程师对字符串池的逻辑做了很大的改变，即将字符串池的位置调整到 Java 堆内，这个改动意味着你再也不会被固定的内存空间限制了。所有的字符串都保存在堆（Heap）中，和其他普通对象一样，这样可以让你在进行调优应用时仅需要调整堆大小就可以了。字符串池概念原本使用得比较多，但是这个改动使得我们有足够的理由让我们重新考虑在 Java 7 中使用 String.intern()。

看一个例子，如代码清单 1-7 所示。

代码清单 1–7 StringIntern 实例程序

```
public class StringInternDemo {
    public static void main(String[] args) {
       String s = new String("1");
       s.intern();
       String s2 = "1";
       System.out.println(s == s2);

       String s3 = new String("1") + new String("1");
       s3.intern();
```

[1] 外文原文：String interning is a method of storing only one copy of each distinct string value, which must be immutable。

```
        String s4 = "11";
        System.out.println(s3 == s4);
    }
}
```

输出结果，JDK6 运行结果为 false false、JDK7 和 JDK8 的运行结果均为 false true。

在 JDK6 中上述的所有打印都是 false 的，如图 1-4 所示，因为 JDK6 中的常量池是放在 Perm 区中的，Perm 区和正常的 Java Heap 区域是完全分开的。上面说过，如果是使用引号声明的字符串都会直接在字符串常量池中生成，而 New 出来的 String 对象是放在 Java Heap 区域。所以拿一个 Java Heap 区域的对象地址和字符串常量池的对象地址进行比较，肯定是不相同的，即使调用 String.intern 方法也是没有任何关系的。

再来看看 JDK7 和 JDK8 的运行结果。这里要明确一点的是，在 JDK6 以及以前的版本中，字符串的常量池是放在堆的 Perm 区的，Perm 区是一个类静态的区域，主要存储一些加载类的信息、常量池、方法片段等内容，默认大小只有 4MB，一旦常量池中大量使用 intern 是会直接产生 Java.lang.OutOfMemoryError: PermGen space 错误的。所以在 JDK7 的版本中，字符串常量池已经从 Perm 区移到正常的 Java Heap 区域了。为什么要移动，Perm 区域太小就是其中一个主要原因。JDK7 代码示例图如图 1-5 所示。

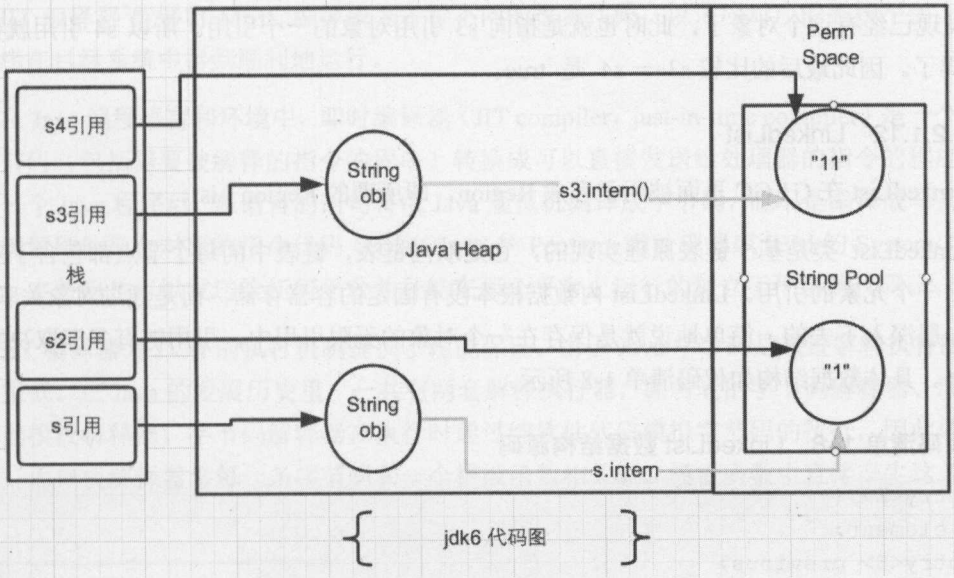


图 1-4　JDK6 代码示例图

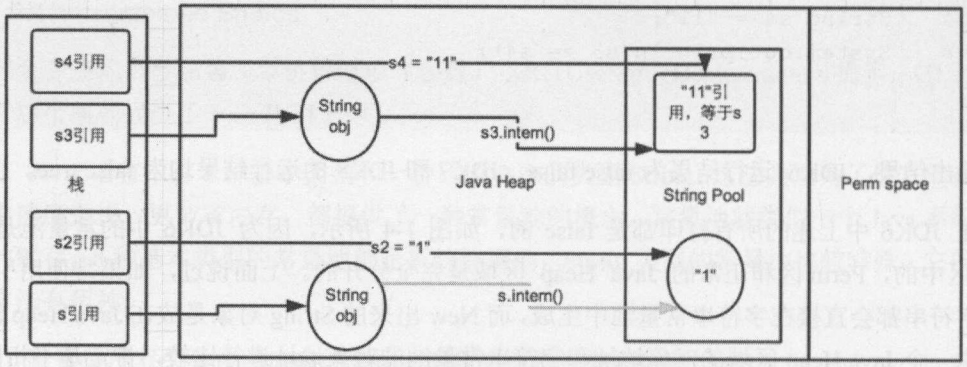


图 1-5　JDK7 代码示例图

先看 s3 和 s4 字符串。String s3 = new String("1") + new String("1");，这句代码中现在生成了 2 个对象，是字符串常量池中的“1” 和 Java Heap 中的 s3 引用指向的对象。接下来 s3.intern(); 这一句代码，是将 s3 中的“11”字符串放入 String 常量池中，因为此时常量池中不存在“11”字符串，因此常规做法是像图 1-4 中表示的那样，在常量池中生成一个 "11" 的对象，关键点是如图 1-5 所示 JDK7 中常量池不在 Perm 区域了，这块做了调整。常量池中不需要再存储一份对象了，可以直接存储堆中的引用。这份引用指向 s3 引用的对象，也就是说引用地址是相同的。最后 String s4 = "11"; 这句代码中"11"是显示声明的，因此会直接去常量池中创建，创建的时候发现已经有这个对象了，此时也就是指向 s3 引用对象的一个引用。所以 s4 引用就指向和 s3 一样了。因此最后的比较 s3 == s4 是 true。

1.2.1.12　LinkedList

LinkedList 在 G1 GC 里面被用于存放 Region，即所谓的 Region List。

LinkedList 类是基于链表原理实现的，它是双向链表，链表中的每个节点都包含了对前一个和后一个元素的引用。LinkedList 内数据根本没有固定的容器存储，而是通过对象关联引用，一层一层深入下去的。简单地说就是保存在一个对象的无限引用中，引用链有多深取决数据量的大小。具体数据结构如代码清单 1-8 所示。

代码清单 1-8　LinkedList 数据结构源码

```
Entry<E> {
E element;
Entry<E> previous;
Entry<E> next;
}
```

LinkedList 实现了 Deque 接口，如果我们需要队列操作，那声明 LinkedList 的实现为 Deque 类型是非常方便的，ArrayList[1]没有实现这个接口。

1.2.1.13　Java 对象头

在 HotSpot 虚拟机中，对象在内存中的布局可以分成对象头、实例数据、对齐填充三部分。

- 对象头：它主要包括对象自身的运行行元数据，比如哈希码、GC 分代年龄、锁状态标志等，同时还包含一个类型指针，指向类元数据，表明该对象所属的类型。
- 实例数据：它是对象真正存储的有效信息，包括程序代码中定义的各种类型的字段（包括从父类继承下来的和本身拥有的字段）。
- 对齐填充：它不是必要存在的，仅仅起着占位符的作用。

对象头大小在 32 位 HotSpot VM 和 64 位 HotSpot VM 之间是不一样的，对象头在 32 位系统上占用 8byte，在 64 位系统上占用 16byte。我们可以通过 Java 对象布局工具获取头大小，这个工具简称为 JOL。

第 4 章会重点涉及 G1 GC 对于 Java 对象头的回收方式介绍。

1.2.1.14　JIT（Just-In-Time）编译器

JIT 编译器[2]能够将 MSIL 编译成为各种不同的机器代码，以适应对应的系统平台，最终使得程序在目标系统中得到顺利地运行。

在 Java 编程语言和环境中，即时编译器（JIT compiler，just-in-time compiler）是一个把 Java 的字节码（包括需要被解释的指令的程序）转换成可以直接发送给处理器的指令的程序。当你写好一个 Java 程序后，源语言的语句将由 Java 虚拟机编译成字节码，而不是编译成与某个特定的处理器硬件平台对应的指令代码（比如，Intel 的 Pentium 微处理器或 IBM 的 System/390 处理器）。字节码是可以发送给任何平台并且能在那个平台上运行的独立于平台的代码。

JIT 编译器为 JVM 的执行机制提供了性能保证。由于 Java 字节码是通过解释执行的，因此效率很低。在 Java 的发展历史里，一共有两套解释执行器，即古老的字节码解释器、现在普遍使用的模板解释器。字节码解释器在执行时通过纯软件代码模拟字节码的执行，因此效率非常低下，而模板解释器将每一条字节码和一个模板函数相关联，模板函数中直接产生这条字节码

---

[1] 一种基于动态数组原理实现的数据结构。

[2] 英文全称是 Just-In-Time Compiler，中文意思是即时编译器。

执行时的机器码[1]，从而很大程度上提高了解释器的性能。但即便如此，仅凭借解释器，JVM 的执行效率依然很低，为了解决这个问题，JVM 平台支持一种叫作即时编译的技术。即时编译的目的是避免函数被解释执行，而是将整个函数体编译成为机器码，每次函数执行时，只执行编译后的机器码即可，这种方式可以使执行效率大幅度提升。

代码清单 1-9　运行 Java 版本及输出

```
C:\Users\Administrator>Java -version
Java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

从代码清单可以看出，目前使用的是混合执行模式（mixed mode）[2]。在混合执行模式中，部分函数会被解释执行，部分可能被编译执行。JVM 决定函数是否需要编译执行的依据是判断该函数是否为热点代码。如果函数的被调用频率很高，那么就是热点，热点代码就会被编译执行。

由于编译执行模式的执行效率会远远高于解释执行模式，所以如代码清单所示，启用编译执行模式。

代码清单 1-10　启用编译模式

```
C:\Users\Administrator>Java -Xcomp -version
Java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, compiled mode)
```

对比一下，运行一个 Java 程序，分别采用-Xint 和-Xcomp 作为 VM 选项，对应的是解释执行模式和编译执行模式。

代码清单 1-11　两种模式对比程序

```
public class JVMDemoTest {
    /**
     * 获取当前 jvm 的内存信息
     * @return
     */
    public static String toMemoryInfo() {
```

[1] 这个原理让我想起了活字印刷术。

[2] JVM 有 3 种执行模式，分别是解释执行、混合执行和编译执行，默认情况是混合执行模式。

```
        int freeMemory;
        int totalMemory;
        String calcResult = null;
        Runtime runtime = Runtime.getRuntime ();
        for(int i = 0;i<100000;i++){
            freeMemory = ( int ) (runtime.freeMemory() / 1024 / 1024);
            totalMemory = ( int ) (runtime.totalMemory() / 1024 / 1024);
            calcResult = freeMemory + "M/" + totalMemory + "M(free/total)";
        }
        return calcResult ;
    }
    /**
     *
     * @param args
     */
    public static void main(String[] args) {
        long b = System.currentTimeMillis();
        System.out.println( "memory info :" + toMemoryInfo ());
        System.out.println(System.currentTimeMillis() - b);
    }
}
```

运行结果显示编译执行模式（耗时 207ms）少于解释执行模式（耗时 726ms）。其实只要是计算较为复杂的程序，都推荐采用编译执行模式。

深入解释一下原理。最早的 Java 编译方案是由一套转译程序将每个 Java 指令都转译成对等的微处理器指令，并根据转译后的指令先后次序依序执行，由于一个 Java 指令可能被转译成十几或数十几个对等的微处理器指令，这种模式执行的速度相当缓慢。针对这个问题，业界首先开发出 JIT（just in time）编译器。当 Java 执行运行时，每遇到一个新的类别（类别是 Java 程序中的功能群组），JIT 编译器在此时就会针对这个类别进行编译（Compile）作业。经过编译后的程序，被优化成相当精简的原生型指令码（Native Code），这种程序的执行速度相当快。花费少许的编译时间来节省稍后相当长的执行时间，JIT 这种设计的确增加不少效率，但是它并未达到最顶尖的效能，因为某些极少执行到的 Java 指令在编译时额外所花费的时间可能比转译器在执行时的时间还长，针对这些指令而言，整体花费的时间并没有减少。基于对 JIT 的经验，业界发展出动态编译器（Dynamic Compiler），动态编译器仅针对较常被执行的程序码进行编译，其余部分仍使用转译程序来执行。也就是说，动态编译器会研判是否要编译某个类别。

JIT 通过智慧机制针对每个类别进行分析，然后决定使用这两种利器的哪一种来达到最佳化的效果。动态编译器针对程序的特性或者是让程序执行几个循环，再根据结果决定是否编译这段

程序码。这个决定不见得绝对正确，但从统计数字来看，这个判断的机制正确的机会相当高。以整个结果来看，动态编译器产生的程序码执行的速度超越以前的 JIT 技术，平均速度可提高 50%。

1.2.1.15 Java Mission Control

JDK 自带了很多工具，JMC 也是其中之一，如图 1-6 所示。

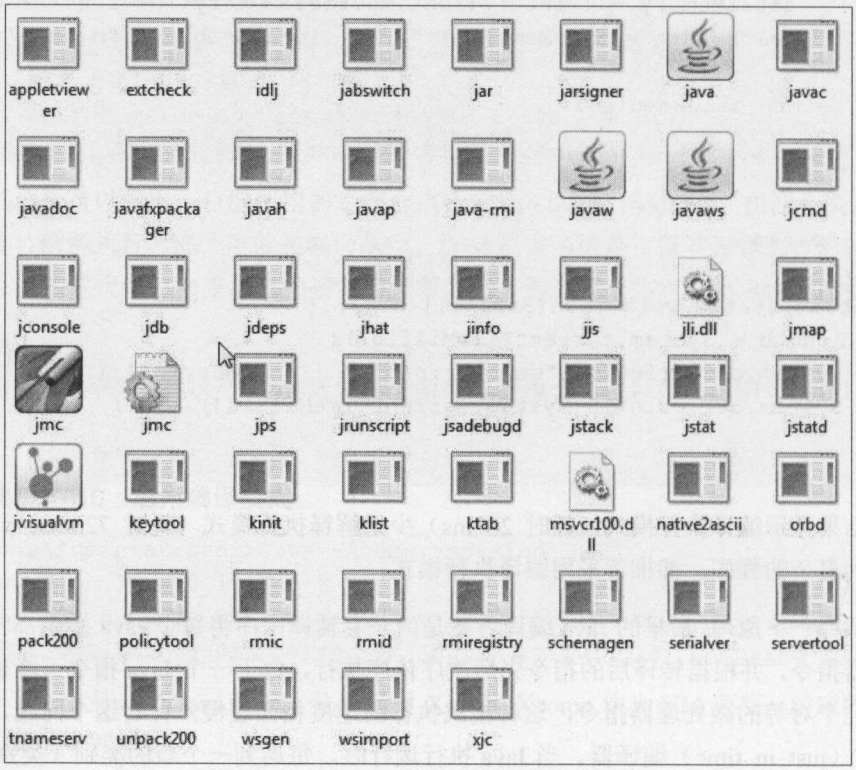


图 1-6 JDK 工具列表图

作为 JVM 融合战略的一部分，主要用来统一 HotSpot、JRockit VMs。目前，JRockit Mission Control 在标准版 Java SE 中已经可以使用。Java Mission Control（JMC）与 Java Flight Recorder 一起工作，适用于 HotSpot JVM，用来记录核心数据和事件。JMC 是一个调优工具，并且适用于 Oracle JDK。一旦出现问题，这些数据就可以用来分析，示例图如图 1-7 所示。

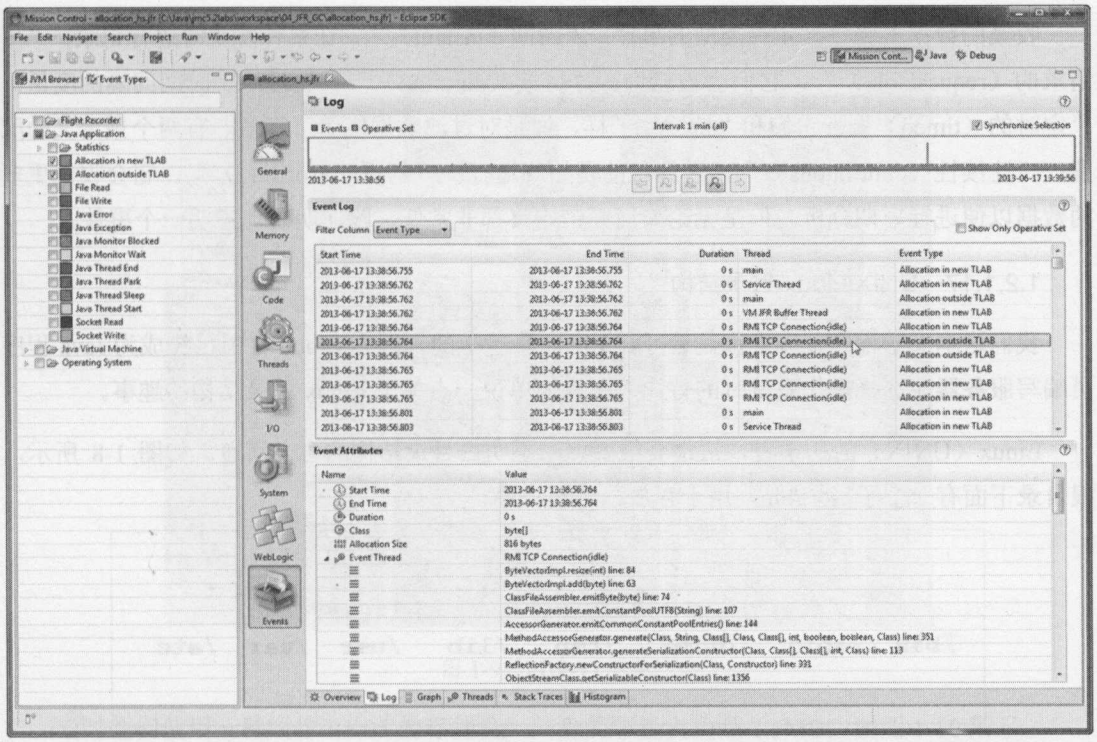


图 1-7　JMC 示例图

1.2.1.16　Java Flight Recorder

第 3 章介绍的最后一个命令行选项-XX:+UnlockCommercialFeatures 会提到 JFR。

如果我们想要在 JVM 之外收集调试数据、特别是在性能数据的工具需要实现 JVMPI/JVMTI 接口的时候。虽然大部分分析工具发展得非常好，但是让它们能够在产品中低消耗地运行还是非常困难的。

Java Flight Recorder 工具支持直接在 JVM 中实现它自己的基于事件的监控接口，所以能够以最小的开销提供 CPU 时间或者对象分配分析这样的功能。例如，这个新接口允许采取线程的样本但不需要它们在还原点上，降低了开销和测量的偏差。只有少量使用字节码检测的事件对运行的代码有影响。大部分捕获技术是新的，第三方无法使用。

Flight Recorder 在 JVM 本地记录数据，但是是记录在堆外（off-heap），因此它并不会影响内存特性和垃圾收集。当它被配置成持久化数据的时候，它会周期性地倾倒（dump）到一个文件中。

收集的数据主要包含 4 种类型的事件：“瞬间（instant）”，在事件发生时进行记录；“可请求的（requestable）”，它们会被轮询；“持续（duration）”，表示一个时间间隔的度量；“定时的（timed）”，它们和“持续”一样，但是对过滤数据应用了阈值。有两个预定义的配置：“连续性（continuous）”，它的目的是始终运行；“剖析（Profiling）”，它会收集更多的数据以便进行短期分析。但是无论如何开销始终都非常低，除非明确地声明一个事件。

1.2.1.17 UNIX/Linux 目录结构

我们这本书的程序基本上都是运行在 Linux 环境上的，因为 Linux 目前已经成为 Java 程序员编写服务器端选择操作系统时的首选，所以简单说一点关于 Linux 目录结构的趣事。

Linux（UNIX）的初学者，常常会很困惑，不明白目录结构的含义何在。如图 1-8 所示，根目录下面有一个子目录/bin，用于存放二进制程序。

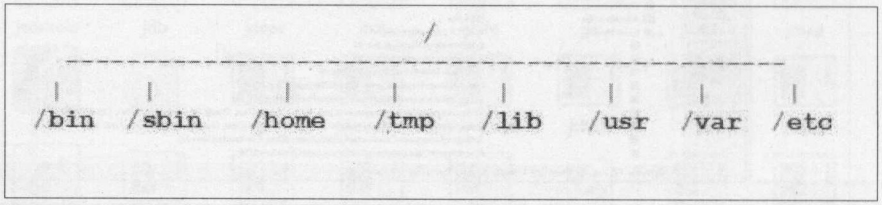


图 1-8 Linux 目录

我们注意到，/usr 子目录下面还有/usr/bin，以及/usr/local/bin，也用于存放二进制程序，某些系统甚至还有/opt/bin，它们有何区别？长久以来，我也感到很费解，不明白为什么这样设计。像大多数人一样，我只是根据“UNIX 文件系统结构标准（Filesystem Hierarchy Standard）”，死记硬背不同目录的区别。直到读到了 Rob Landley 的简短解释，这才恍然大悟，原来 UNIX 目录结构是历史造成的。

话说 1969 年，Ken Thompson 和 Dennis Ritchie 在小型机 PDP-7 上发明了 UNIX。1971 年，他们将主机升级到了 PDP-11，如图 1-9 所示。

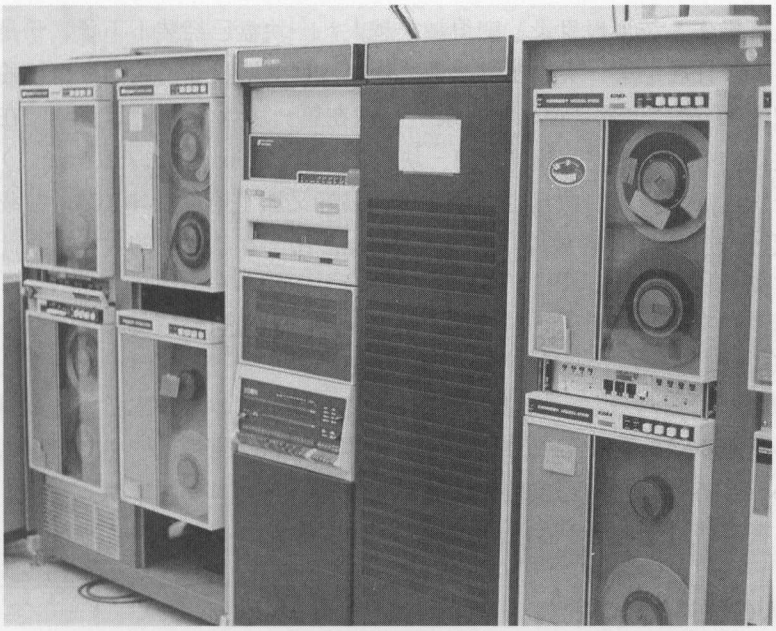


图 1-9　PDP-11 主机

当时他们使用一种叫作 RK05 的存储盘，一盘的容量大约是 1.5MB，如图 1-10 所示。

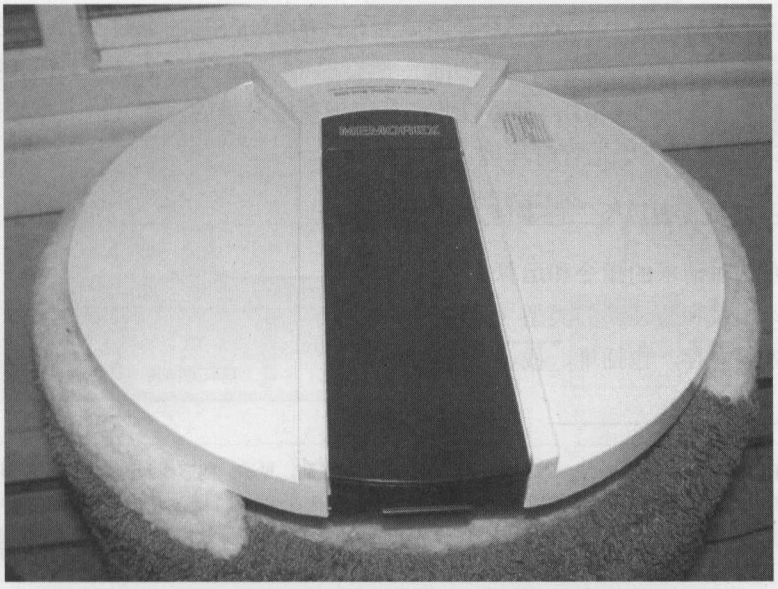


图 1-10　RK05 存储盘

没过多久，操作系统（根目录）变得越来越大，一块盘已经装不下了。于是，他们加上了第二盘 RK05，并且规定第一块盘专门放系统程序，第二块盘专门放用户自己的程序，因此挂载的目录点取名为/usr。也就是说，根目录"/"挂载在第一块盘，"/usr"目录挂载在第二块盘。除此之外，两块盘的目录结构完全相同，第一块盘的目录（/bin, /sbin, /lib, /tmp...）都在/usr 目录下重新出现一次。后来，第二块盘也满了，他们只好又加了第三盘 RK05，挂载的目录点取名为/home，并且规定/usr 用于存放用户的程序，/home 用于存放用户的数据。从此，这种目录结构就延续了下来。

随着硬盘容量越来越大，各个目录的含义进一步得到明确。

- /：存放系统程序，也就是 AT&t 开发的 UNIX 程序。
- /usr：存放 UNIX 系统商（比如 IBM 和 HP）开发的程序。
- /usr/local：存放用户自己安装的程序。
- /opt：在某些系统，用于存放第三方厂商开发的程序，所以取名为 option。

1.2.2 JVM/GC 通用术语

1.2.2.1 HotSpot VM

Java HotSpot 虚拟机最初是由 Longview/Animorphic 公司实现的[1]，随着公司被 Sun 公司收购而成为 Sun 的 JVM，并于 JDK 1.3.0 开始成为 Sun 的 Java SE 的主要 JVM。在 Sun[2]被 Oracle 公司[3]收购后，HotSpot VM 逐步取代了 Oracle 原有的 JRockit JVM[4]，成为 Java SE 的主要 JVM。Oracle 的 JDK 和开源的 OpenJDK 都是以此虚拟机为基础发展的。和其他虚拟机一样，HotSpot 虚拟机为字节码提供了一个运行时环境。

HotSpot 主要负责做以下三件事情。

- 执行方法所请求的指令和运算。
- 定位、加载和验证新的类型（即类加载）。
- 管理应用内存，包括堆、栈、方法区等。

---

[1] 参见 Java History，http://c2.com/cgi/wiki?JavaHistory，格式有点乱，见谅，能够找到的就这么点资料了。

[2] 一家传奇硅谷公司，成立第一个季度就开始赢利，但是无法创造持续的赢利点，造成最后被甲骨文收购。

[3] 即甲骨文股份有限公司（甲骨文软件系统有限公司），是全球最大的企业级软件公司，总部位于美国加利福尼亚州的红木滩。

[4] 事实上，Oracle JRockit（原来的 Bea JRockit）系列产品是一个全面的 Java 运行时解决方案组合，包括了行业最快的标准 Java 解决方案。大量的行业基准测试显示，基本上 JRockit JVM 是世界上最快的 JVM。

我们需要知道，HotSpot 是一个混合执行模式的虚拟机，也就是说它既可以解释字节码，又可以将代码编译为本地机器码，这样执行起来就会更快。

HotSpot 虚拟机可以运行在两种模式下，即 Client 和 Server 模式。你可以在 JVM 启动时通过配置-Client 或者-Server 选项来选择其中一种。两种模式最主要的区别是，Server 模式下会进行更激进的优化措施，在 Server 模式下，HotSpot 虚拟机会默认在解释模式下运行方法 10000 次才会触发 JIT 编译。

HotSpot VM 有一个稳定强悍的架构，支持强大的功能与特性，具备实现高性能和大规模可伸缩性的能力，例如 JIT 编译器能动态进行优化生成。换句话说，它们运行 Java 程序时会针对底层系统架构动态生成高性能的本地机器指令。此外，通过针对运行时环境的不断改进、稳定的版本发布，加上不断改进和发布的不同种类的多线程垃圾收集器，HotSpot VM 即使是在大型计算机系统上也能获得很高的伸缩性。

如图 1-11 所示，HotSpot JVM 由类加载子系统、运行时数据区、执行引擎、方法接口等组成。

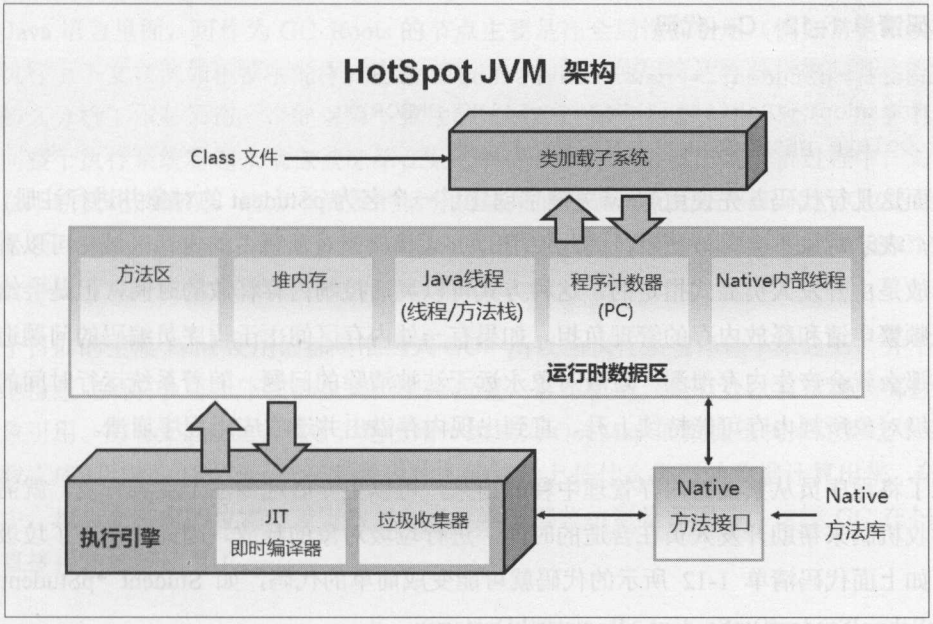


图 1-11　HotSpot JVM 架构图

1.2.2.2　垃圾回收（Garbage Collection）

计算机系统体系对我们来说越来越远，在不了解底层实现方式的前提下，通过高级语言很

容易编写程序代码。但事实上计算机并不认识高级语言，在执行过程中我们会把高级语言转换成计算机所能理解的一种中间格式（如汇编语言），然后才能理解计算机如何解释和执行这些中间的程序，以及系统的哪一部分影响程序的执行效率。因此，作为一门高级语言，Java 必须提供垃圾回收机制来确保它自身不会发生异常。

对于高级语言来说，一个基本认知是如果不进行垃圾回收，内存迟早都会被消耗完，因为不断地分配内存空间而不进行回收，就好像不停地生产生活垃圾而从来不打扫一样。

请注意，垃圾回收机制并不是 Java 虚拟机独创的，早在上世纪 60 年代，垃圾回收就已经被 Lisp 语言[1]所使用。现在，除了 Java 以外，C#、Python 等语言也都使用了垃圾回收的思想。可以说，这种自动化的内存管理方式已经成为现代开发语言必备的标准了。

在早期的 C++语言时代，垃圾回收基本上是手工进行的。开发人员可以使用 New 关键字进行内存申请，并使用 Delete 关键字进行内存释放。比如代码清单 1-12 所示。

代码清单 1-12　C++代码

```
Student *pStudent = new cmBaseSchoolStudent();
If(pStudent->Register(kDestory)!=NO_ERROR)
   Delete pStudent;
```

上面这几行代码首先使用 New 关键字创建了一个名为 pStudent 的对象并进行注册，如果注册失败，表示对象不会再被使用，因此使用 Delete 释放该对象所占的内存区域。可以看到，内存的释放是由开发人员显式指定的，这种方式可以灵活控制内存释放的时间，但是会给开发人员带来频繁申请和释放内存的管理负担。如果有一处内存区间由于程序员编码的问题造成忘记回收，那么就会产生内存泄漏，垃圾对象永远无法被清除的问题，随着系统运行时间的不断增长，垃圾对象所耗内存可能持续上升，直到出现内存溢出并造成应用程序崩溃。

为了将程序员从繁重的内存管理中释放出来，可以更专心地专注于业务开发，就需要一个垃圾回收机制来帮助开发人员在合适的时间，进行垃圾对象的释放。因此，在有了垃圾回收机制后，如上面代码清单 1-12 所示的代码就可能变成简单的代码，如 Student *pStudent = new cmBaseSchoolStudent();pStudent->Register(kDestory);。

回到 Java。垃圾收集是 Java 语言非常显著的特点，不像 C 语言那样，老是要考虑什么数字

[1] 一种通用高级计算机程序语言，长期以来垄断人工智能领域的应用。

越界这样类似的问题。在 Java 语言里，什么是垃圾（Garbage）[1]呢？垃圾是指在运行程序中没有任何指针指向的对象，这个对象就是需要被回收的垃圾。

通俗点讲，在 Java 中，当没有对象引用指向原先分配给某个对象的内存时，该内存便成为垃圾。JVM 的一个系统级线程会自动释放该内存块，垃圾意味着程序不再需要的对象是“无用信息”，这些信息会被丢弃。当一个对象不再被引用的时候，内存回收它占领的空间，以便被后来的新对象使用，这个回收过程可能是在年轻代回收，也可能是在老年代回收，甚至是 Full GC 阶段。

事实上，除了释放没用的对象，垃圾回收也可以清除内存里的记录碎片。由于创建对象和垃圾回收器释放丢弃对象所占的内存空间时，内存会因此出现碎片。碎片是分配给对象的内存块之间的空闲内存，碎片整理将所占用的堆内存移到堆的一端，以便 JVM 将整理出的内存分配给新的对象。

1.2.2.3　枚举根节点

在 Java 语言里面，可作为 GC Roots 的节点主要是在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中。如果要使用可达性分析来判断内存是否可回收，那么分析工作必须在一个能保障一致性的快照中进行——这里“一致性”的意思是整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中，对象引用关系还在不断变化的情况，这点不满足的话分析结果的准确性就无法保证。这点也是导致 GC 进行时必须“Stop the world”的一个重要原因，即使是号称（几乎）不会发生停顿的 CMS 收集器中，枚举根节点时也是必须要停顿的。

由于目前的主流 JVM 使用的都是准确式 GC，所以当执行系统停顿下来之后，并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机应当是有办法直接得到哪些地方存放着对象引用。在 HotSpot 的实现中，是使用一组成为 OopMap 的数据结构来达到这个目的的，在类加载完成的时候，HotSpot 就把对象内什么偏移量上是什么类型的数据计算出来，在 JIT 编译过程中，也会在特定的位置记录下栈里和寄存器里哪些位置是引用的。这样 GC 在扫描时就就可以直接得知这些信息了。

---

[1] 国外书籍对于垃圾的定义：An object is considered garbage when it can no longer be reached from any pointer in the running program。

1.2.2.4 并行（Parallellism）

并行，在软件领域通常指的是计算机系统中能同时执行两个或多个处理的一种计算方法。并行处理的主要目的是节省大型和复杂问题的解决时间。从理论上讲，*N* 个并行处理的执行速度可能会是在单一处理机上执行速度的 *N* 倍。

在 GC 内部，并行意味着多线程执行 GC 操作，即多个垃圾线程同时并行执行，用户线程处于等待状态。在 HotSpot 垃圾收集器里，除了 G1 以外，其他的垃圾收集器使用内置的 Java VM（JVM）线程执行 GC 的多线程操作，而 G1 GC 采用的是应用线程承担后台运行的 GC 工作，即当 VM 的线程处理速度慢时，系统会调用应用程序线程帮助加速垃圾回收过程。

有另一个和并行很相似的名词，但是实际意义不同，就是并发（Concurrency）。并发和并行的差别主要是以下几点。

- 并行是指两个或者多个事件在同一时刻发生，而并发是指两个或多个事件在同一时间间隔发生。
- 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。
- 并行是在一台处理器上“同时”处理多个任务，并发是在多台处理器上同时处理多个任务。如 Hadoop 分布式集群。

并发在 JVM 里面，是指垃圾收集线程和用户线程同时执行，但不一定是并行执行，可能是交叉执行，用户程序继续运行，而垃圾收集程序运行在另一个 CPU 上。

1.2.2.5 吞吐量（Throughput）

吞吐量主要关注一个特定时间段内应用系统的最大工作量。衡量吞吐量的指标包括以下内容。

- 给定时间内完成的事务数。
- 每小时批处理系统能完成的作业（jobs）数量。
- 每小时能完成多少次数据库查询。

在吞吐量方面优化的系统，停顿时间长（High Pause Times）也是可以接受的。由于高吞吐量应用运行时间长，所以此时更关心的是如何尽可能快地完成整个任务，而不考虑快速响应。

大家都知道 GC 暂停很容易造成性能瓶颈。现代 JVM 在发布的时候都自带了高级的垃圾回收器，不过从我的使用经验来看，要找出某个应用最优的配置真是难上加难[1]。手动调优是必需

[1] 今年在 CSDN 上做了一次问答活动，不少人问有没有一套 GC 参数调优默认值，我说没有，因为我不了解你的应用程序运行业务逻辑、设计思维，给出一套所谓指导性意见，那不是技术人员应该做的。

的，但是你得了解 GC 算法的确切机制才行。

用来演示 GC 对吞吐量产生影响的应用只是一个简单的程序，它包含以下两个线程。

- PigEater：它会模仿巨蟒不停吞食大肥猪的过程。代码是通过往 Java.util.List 中添加 32MB 字节来实现这点的，每次吞食完后会睡眠 100ms。
- PigDigester：它模拟异步消化的过程。实现消化的代码只是将猪的列表置为空。由于这是个很累的过程，因此每次清除完引用后这个线程都会睡眠 2000ms。

两个线程都会在一个 while 循环中运行，不停地吃了再消化直到蛇吃饱为止。这大概得吃掉 5000 头猪。

代码清单 1-13　吞吐量示例

```
public class PigInThePython {
  static volatile List pigs = new ArrayList();
  static volatile int pigsEaten = 0;
  static final int ENOUGH_PIGS = 5000;

  public static void main(String[] args) throws InterruptedException {
    new PigEater().start();
    new PigDigester().start();
  }

  static class PigEater extends Thread {

    @Override
    public void run() {
      while (true) {
        pigs.add(new byte[32 * 1024 * 1024]); //32MB per pig
        if (pigsEaten > ENOUGH_PIGS) return;
        takeANap(100);
      }
    }
  }

  static class PigDigester extends Thread {
    @Override
    public void run() {
      long start = System.currentTimeMillis();
```

```
        while (true) {
          takeANap(2000);
          pigsEaten+=pigs.size();
          pigs = new ArrayList();
          if (pigsEaten > ENOUGH_PIGS)  {
            System.out.format("Digested %d pigs in %d ms.%n",pigsEaten,
System.currentTimeMillis()-start);
            return;
          }
        }
      }
    }

    static void takeANap(int ms) {
      try {
        Thread.sleep(ms);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

现在我们将这个系统的吞吐量定义为“每秒可以消化的猪的头数”。考虑到每 100ms 就会有猪被塞到这条蟒蛇里，可以看到这个系统理论上的最大吞吐量可以达到 10 头/秒。

来看下使用两个不同的配置系统的表现分别是什么样的。不管是哪个配置，应用都运行在一台拥有双核 CPU，8GB 内存的机器上。

- 第一个配置是 4G 的堆（-Xms4g -Xmx4g），使用 CMS 来清理老年代（-XX:+UseConcMarkSweepGC），使用并行回收器清理新生代（-XX:+UseParNewGC），将堆的 12.5%（-Xmn512m）分配给新生代，并将 Eden 区和 Survivor 区的大小限制为一样的。
- 第二个配置则略有不同，使用 2G 的堆（-Xms2g -Xms2g），新生代和老年代都使用 Parellel GC(-XX:+UseParallelGC)，将堆的 75%分配给新生代（-Xmn 1536m）。

哪个配置的表现会更好一些（就是每秒能吃多少猪），来看一下结果。

- 第一个配置（大堆，大的老年代，CMS GC）每秒能吞食 8.2 头猪。
- 第二个配置（小堆，大的新生代，Parellel GC）每秒可以吞食 9.2 头猪。

现在来客观地看待一下这个结果。分配的资源少了 1/2 但吞吐量提升了 12%。这和常识正好相反，因此有必要进一步分析下到底发生了什么。选择的不同结果也会对吞吐量和容量规划

产生很大的影响。

1.2.2.6　堆内存快照（Java Heap Dump）

开发应用程序过程中经常会遇到 OutOfMemory 异常，而且常常是过一段时间内存才被吃光，这里可以利用 Java Heap Dump 出 JVM 的内存镜像，然后再对其进行分析来查找问题。

Java Heap 是分配给实例类和数组对象运行的数据区，所有 Java 线程在运行期间共享 Heap 中的数据。Java Heap Dump 相当于 Java 应用在运行的时候在某个时间点上打了个快照(SnapShot)。

有以下方法[1]触发 Heap Dump。

- 使用$JAVA_HOME/bin/jmap –dump 命令来触发，JMap 是 JDK 自带的一个调试程序，例如 jmap -dump:format=b,file=/home/longhao/heamdump.out;。
- 使用$JAVA_HOME/bin/jcosole 这个 JDK 自动调试工具的 MBean 子功能，通过选择 MBean>com.sun.management>HotSpotDiagnostic>操作>dumpHeap，点击 dumpHeap 按钮，生成的 Dump 文件在 Java 应用的根目录下面。
- 在应用启动时配置相关的选项-XX:+HeapDumpOnOutOfMemoryError，当应用抛出 OutOfMemoryError 时生成 Dump 文件。
- 使用 hprof。启动虚拟机加入-Xrunhprof:head=site，会生成 Java.hprof.txt 文件。该配置会导致 JVM 运行非常慢，不适合生产环境。

我们举一个例子，比如第一种方法，jmap 命令输出如代码清单 1-14 所示。

代码清单 1-14　jmap 命令输出

```
 Attaching to process ID ****, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 1.8.0_101

using thread-local object allocation.
Mark Sweep Compact GC

Heap Configuration:
   MinHeapFreeRatio  = 40
```

[1] 本书第 6 章重点介绍 JDK 自带的 JVM 调试工具 Serviceability Agent。

```
   MaxHeapFreeRatio  = 70
   MaxHeapSize       = 67108864 (64.0MB)
   NewSize           = 655360 (0.625MB)
   MaxNewSize        = 4294901760 (4095.9375MB)
   OldSize           = 1441792 (1.375MB)
   NewRatio          = 12
   SurvivorRatio     = 8
   PermSize          = 8388608 (8.0MB)
   MaxPermSize       = 67108864 (64.0MB)

Heap Usage:
New Generation (Eden + 1 Survivor Space):
   capacity  = 4521984 (4.3125MB)
   used      = 1510200 (1.4402389526367188MB)
   free      = 3011784 (2.8722610473632812MB)
   33.39684527853261% used
Eden Space:
   capacity  = 4063232 (3.875MB)
   used      = 1495992 (1.4266891479492188MB)
   free      = 2567240 (2.4483108520507812MB)
   36.81778446320565% used
From Space:
   capacity  = 458752 (0.4375MB)
   used      = 14208 (0.0135498046875MB)
   free      = 444544 (0.4239501953125MB)
   3.09709821428571448 used
To Space:
   capacity  = 458752 (0.4375MB)
   used      = 0 (0.0MB)
   free      = 458752 (0.4375MB)
   0.0% used
tenured generation:
   capacity  = 59342848 (56.59375MB)
   used      = 36321192 (34.638587951660156MB)
   free      = 23021656 (21.955162048339844MB)
   61.20567721994064% used
Perm Generation:
   capacity  = 11796480 (11.25MB)
   used      = 11712040 (11.169471740722656MB)
   free      = 84440 (0.08052825927734375MB)
   99.28419325086806% used
```

以上的输出很简单，第4行起开始输出此进程我们的JAVA使用的环境。Heap Configuration，指在我们启动时设置的一些JVM选项，例如最大使用内存大小，老年代、年轻代、持久代大小等。

1.2.2.7　根集合（Root Set）

根集合会涉及堆和栈的相关知识，先介绍一下。

堆和栈是程序运行的关键，区别如下（见图 1-12）所示。

- 栈是运行时的单位，而堆是存储的单位。
- 栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

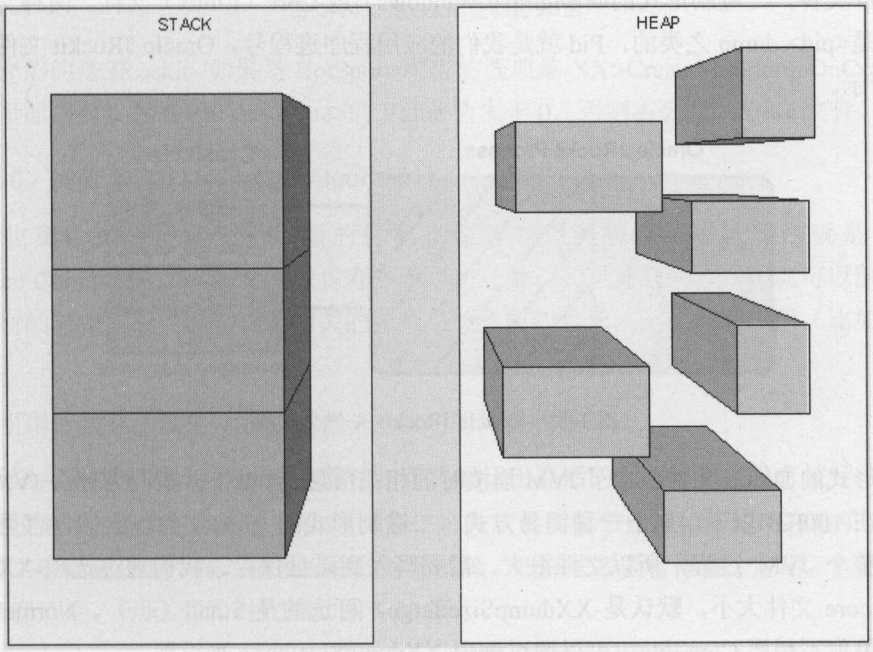


图 1-12　堆和栈区分图

在 Java 中有一个线程就会相应地有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈，而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关的信息。包括局部变量、程序运行状态、方法返回值等等，而堆只负责存储对象信息。

根集合[1]在每一代 GC 里都有提及，简单地讲就是栈里面的对象引用和堆内存里面的对象的集合。从这些对象开始就可以一级一级地追踪到所有存活对象。第 4 章会提及标记/清除阶段，其中的标记阶段就是从 Root 开始扫描，由于 Root 采用栈方式存放变量和指针，所以如果一个指针，它保存了堆内存里面的对象，但是自己又不存放在堆内存里面，那它就是一个 Root。Root 是一个全局变量，GC 通过它来判别哪些对象需要回收，哪些对象依然是存活的。

1.2.2.8 崩溃文件（Crash Dump Core Files）

默认情况下所有的操作系统都会在操作系统状态发生变化，或者说发生了异常时，会生成系统级别的崩溃日志，对应的 JVM 也创造了类似的机制。

一般情况下，JVM[2]会创建两种类型的崩溃文件，即文本形式的、二进制形式的。文本形式的叫 Dump 文件，二进制形式的叫 Mdmp（Windows）或 Core（Linux）文件。两种文件的文件名一般都是<pid>.dump 之类的，Pid 就是我们的应用程序进程号。Oracle JRockit 文件生成图如图 1-13 所示。

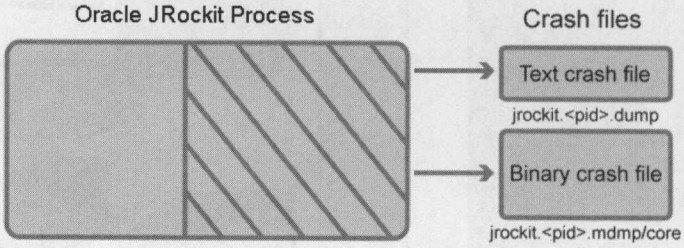


图 1-13　Oracle JRockit 文件生成图

文本形式的 Dump 文件包含了 JVM 崩溃时的相关信息，可以告诉我们为什么 JVM 崩溃了，一般控制在 100KB 以下，属于一种简易方式。二进制形式的 Core 文件包含的内容更多了，一般情况是整个 JVM 过程，所以文件很大，需要写入到磁盘保存。我们通过选项-XXdumpSize 可以设置 core 文件大小，默认是-XXdumpSize:large，可选的是 Small（小）、Normal（正常）模式。如果你不想要 Core File，可以通过选项-XXdumpSize:none 来设置。

图 1-14 总结了三种模式的差异点，覆盖范围一目了然。

---

[1] Oracle 原文定义为 Any object referred to by a root is reachable and is therefore a live object. Additionally, any objects referred to by a live object are also reachable。

[2] 这里特指 Oracle JRockit JVM。

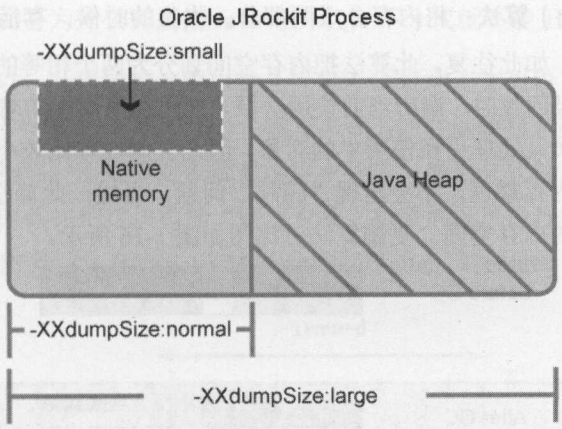


图 1-14　JRockit 进程分配图

前面介绍的是 JRockit，如果是 HotSpot，对应的选项是-XX:-CreateMinidumpOnCrash。Linux 操作系统里面需要设置 ulimit -c <value>的 Value 值大于 0，否则不会写入 Core 文件。

1.2.2.9　回收算法（Collection Algorithm）

垃圾收集针对的是堆内存里面的对象。那首先想到的最简单的算法就是引用计数（Reference Counting），对每个对象保存一个引用计数，如果计数为零，那就可以删除了。但是缺点是新的对象生成就需要更新相关的计数，更重要的是无法删除循环引用（比如两个对象彼此引用）。

除了引用计数算法以外，我们还有一些常用的垃圾收集算法。

- **标记—清除（Mark-Sweep）算法：**首先标记出存活的对象，那些没有被标记的就可以被收集了。此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。标记-清除对比图如图 1-15 所示。

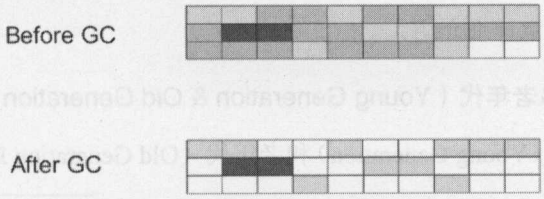


图 1-15　标记-清除对比图

- **复制（Copying）算法：**将内存分成两部分，收集的时候，存活的对象从一部分被移动到另一个部分，如此往复。此算法把内存空间划分为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍的内存空间。复制算法对比图如图 1-16 所示。

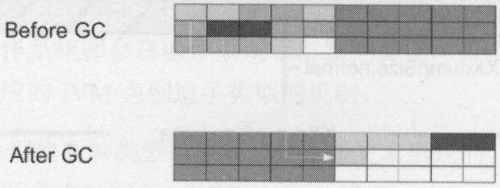


图 1-16　复制算法对比图

- **标记–整理（Mark–Compact）算法：**此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。标记-整理算法对比图如图 1-17 所示。

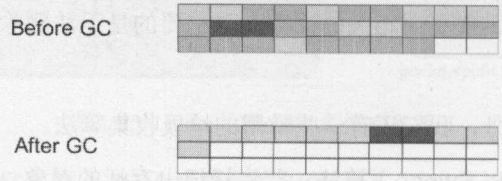


图 1-17　标记-整理算法对比图

- **分代收集（Generational Collecting）算法：**这个也是 Java 使用的算法，基于统计结果并综合了前面算法优点。

具体内容会在第 2 章详细介绍。

1.2.2.10　年轻代&老年代（Young Generation & Old Generation）

为什么会有年轻代（Young Generation）和老年代[1]（Old Generation 或 Tenured Generation）？

[1] 很多文章也翻译为年老代，我觉得还是叫老年代顺口，所以本书都叫老年代。

回答这个问题之前先来回答为什么需要把 Java 堆分代？不分代就不能正常工作了吗？[1]

其实不分代完全可以，分代的唯一理由就是优化 GC 性能。如果没有分代，那所有的对象都在一块，就如同把一个学校的人都关在一个教室。GC 的时候要找到哪些对象没用，这样就会对堆的所有区域进行扫描。而很多对象都是朝生夕死的，如果分代的话，把新创建的对象放到某一地方，当 GC 的时候先把这块存储“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。如图 1-18 所示，HotSpot VM 被分成了 Young、Tenured、Perm 三个阶段。

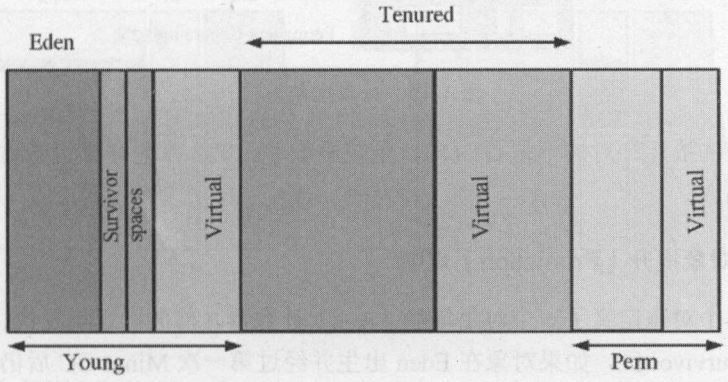


图 1-18　HotSpot 分代图

HotSpot VM 把年轻代分为了 3 部分，即 1 个 Eden 区和 2 个 Survivor 区（分别叫 From 和 To），默认比例为 8∶1。当 GC 只发生在年轻代中，回收年轻代对象的行为被称为 Minor GC。当 GC 发生在老年代时则被称为 Major GC 或者 Full GC。一般的，Minor GC 的发生频率要比 Major GC 高很多，即老年代中垃圾回收发生的频率将大大低于年轻代。

整个生命周期如图 1-19 所示。

[1] 这个问题国外有文章说和 infant mortality 类似，单词大意就是很多对象的生存时间很短，很少的对象会存活很久。因此堆被分成了几部分，分别存放不同寿命的对象。

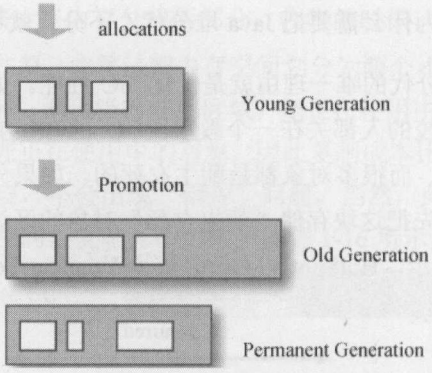


图 1-19　对象生命周期

注意，以上的描述针对的不是 G1 GC，是较早版本。第 3 章会详细讨论这类 GC 日志输出后的结果分析问题。

1.2.2.11　对象提升（Promotion）规则

虚拟机给每个对象定义了一个对象年龄（Age）计数器。前面看过年轻代、老年代的介绍，了解了 Eden、Survivor 区，如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）[1]时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过选项-XX:MaxTenuringThreshold 来设置。

针对不同年龄段的对象分配原则如下所示。

1. 对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC。

2. 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象），G1 GC 针对大对象有自己的处理方法，请看第 4 章。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（年轻代采用复制算法收集内存）。

3. 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 会进入 Survivor 区，之后每经过一次 Minor GC，则对象的年龄加 1，直到达到阀值对象进入老年区。

[1] 这里只是通俗讲解，其实每个 JVM 都不一样，每个 GC 也不一样。

4. 动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

5. 空间分配担保。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC，如果小于则进入检查 HandlePromotionFailure 逻辑。判断这个逻辑，如果是 True 则只进行 Minor GC，如果是 False 则进行 Full GC，具体逻辑实现如图 1-20 所示。

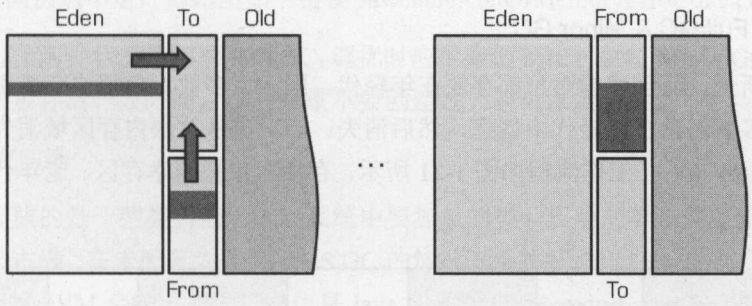


图 1-20　Eden 移动数据示意图

1.2.2.12　永久区（PermGen Space）

PermGen Space 的全称是 Permanent Generation Space，是指内存的永久保存区域。这一部分用于存放 Class 和 Meta 的信息，Class 在被加载的时候被放入 PermGen Space 区域。它和存放常量的堆区域不同，GC 不会在主程序运行期对 PermGen Space 进行清理，所以如果应用程序会加载很多类的话，就很可能出现 PermGen Space 错误。这种错误常见在 WEB 服务器对 JSP 进行预先编译的时候，如果你的 Web App 使用了大量的第三方 Jar 包，并且其大小超过了 JVM 默认的大小，那么就会产生此错误信息了。

在整个 JDK7 的更新过程当中[1]，永久区还是存在并被使用的，只不过已经开始从它内部移出数据了，大致一共移出了三类数据：

- 例如&APPLID[2]这样的符号（Symbols）被移到了本地堆区；
- 内部字符串被移到了 Java 堆区；
- 类静态属性被移到了 Java 堆区。

---

[1] 一般来说，一个大版本的 JDK 会有持续 1.5~2 年的更新周期，都以小版本命名，比如 jdk7u45。

[2] 如果使用这个标记，CICS 区域的 APPLTD 会在运行时被替换。

如果你使用的是 JDK7，而 GC 使用的是 G1 GC，那么永久区只有在 Full GC 阶段才会被回收。G1 只会在永久区满了后才调用 Full GC 事件，或者在应用程序的生产速度比 G1 的垃圾回收速度快时调用。

JDK8 HotSpot VM 开始使用本地化的内存空间来存放类的元数据，这个空间叫作元空间（Metaspace）。这样的修改意味着 Java.lang.OutOfMemoryError: PermGen 的空间问题将不复存在，并且不再需要调整和监控这个内存空间，也就是说，在 JDK8 完成了对永久区的移除[1]。

1.2.2.13 Full GC & Minor GC

正如前面所说，新创建的对象都存放在年轻代。因为大多数对象很快变成引用不可达（死亡），所以大多数对象在年轻代中创建，然后消失。当对象从这块内存区域消失时，我们说发生了一次“Minor GC”，工作流程如图 1-21 所示，存活对象根据幸存区、老年代区间的实际大小进行调整。

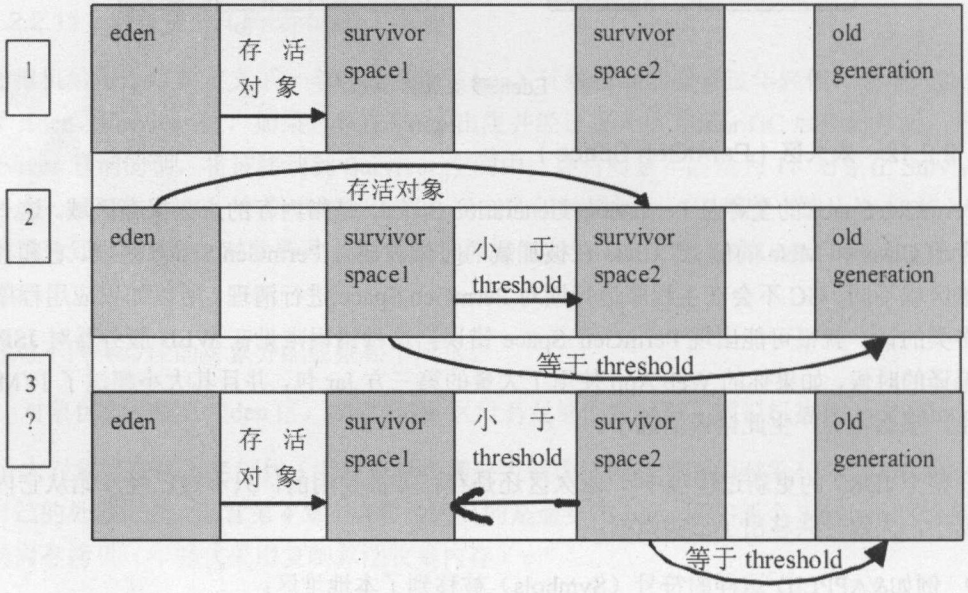


图 1-21 Minor GC 工作原理图

存活下来的年轻代对象被复制到老年代。老年代的内存区域一般大于年轻代。因为它拥有

[1] 移除是基于这个缺陷，https://bugs.openjdk.Java.net/browse/JDK-6964458。

更大的规模，为了提高系统整体性能，所以 GC 发生的次数比在年轻代的少。对象从老年代消失时，我们说“Major GC”或“Full GC”发生了，就是魔王放大招了。

除了直接调用 System.gc 外，触发 Full GC 执行的情况有如下四种。

1. 老年代空间不足

老年代空间只有在年轻代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行 Full GC 后空间仍然不足，则抛出如下错误 Java.lang.OutOfMemoryError: Java heap space。

为了避免以上两种状况引起的 Full GC，调优时应尽量做到让对象在 Minor GC 阶段被回收，让对象在年轻代多存活一段时间，以及尽量不要创建过大的对象及数组。

2. 永久代空间满

永久代中存放的是一些类的信息，当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下会执行 Full GC。如果经过 Full GC 仍然回收不了，那么 JVM 会抛出如下错误信息 Java.lang.OutOfMemoryError: PermGen space。

为了避免永久代被占满造成 Full GC 现象，可采用的方法为增大永久代空间或转为使用 CMS GC。

3. CMS GC 时出现 Promotion Failed 和 Concurrent Mode Failure

对于采用 CMS 进行老年代 GC 的程序而言，尤其要注意 GC 日志中是否有 Promotion Failed 和 Concurrent Mode Failure 两种状况，当这两种状况出现时可能会触发 Full GC。Promotion Failed 是在进行 Minor GC 时，Survivor Space 放不下、对象只能放入老年代，而此时老年代也放不下时造成的[1]。Concurrent Mode Failure 是在执行 CMS GC 的过程中，同时有对象要放入老年代，而此时老年代空间不足造成的。

应对措施为增大 Survivor Space、老年代空间或调低触发并发 GC 的比率，但在 JDK5.0+、6.0+的版本中有可能会由于 JDK 的 Bug29 导致 CMS 在 Remark 完毕后很久才触发清除动作。对于这种状况，可通过设置选项-XX:CMSMaxAbortablePrecleanTime=5（单位为毫秒）来避免。

4. 统计得到的 Minor GC 晋升到老年代的平均大小大于老年代的剩余空间

这是一个较为复杂的触发情况，HotSpot 为了避免由于年轻代对象晋升到老年代导致老年代

[1] 这里描述较为口语化，简单地说就是哪里都放不下了，只能 Full GC 干活了。

空间不足的现象，在进行 Minor GC 时，做了一个判断，如果之前统计所得到的 Minor GC 晋升到老年代的平均大小大于老年代的剩余空间，那么就直接触发 Full GC。

例如程序第一次触发 MinorGC 后，有 6MB 的对象晋升到老年代，那么当下一次 Minor GC 发生时，首先检查老年代的剩余空间是否大于 6MB，如果小于 6MB，则执行 Full GC。当年轻代采用 Parallel GC 时，方式稍有不同，Parallel GC 是在 Minor GC 后也会检查，例如上面的例子中第一次 Minor GC 后，GC 会检查此时老年代的剩余空间是否大于 6MB，如小于，则触发对老年代的回收。

除了以上 4 种状况外，对于使用 RMI 来进行 RPC 或管理的 Sun JDK 应用而言，默认情况下会一小时执行一次 Full GC，这个执行间隔时间是可以配置的。允许在启动时通过-Java-Dsun.rmi.dgc.client.gcInterval=3600000 来设置 Full GC 执行的间隔时间或通过-XX:+DisableExplicitGC 来禁止 RMI 调用 System.gc。

1.2.2.14 Stop the World

看到这个单词的时候我忽然想起《灌篮高手》[1]的一幕，樱木花道[2]说要统治全宇宙，被赤木刚宪[3]一记闷拳，“傻瓜，你只能统治篮球场的两端”。Stop-the-World，简称 STW，指的是 GC 事件/过程[4]发生过程当中停止所有的应用程序线程的执行。这让我想起了我丈母娘对我喊着：“我扫垃圾的时候你站在原地不要动”。

垃圾回收器的任务是识别和回收垃圾对象进行内存清理。为了让垃圾回收器可以正常且高效地执行，大部分情况下会要求系统进入一个停顿的状态。停顿的目的是终止所有应用程序的执行，只有这样，系统中才不会有新的垃圾产生，同时停顿保证了系统状态在某一个瞬间的一致性，也有益于垃圾回收器更好地标记垃圾对象。因此，在垃圾回收时，都会产生应用程序的停顿。停顿产生时整个应用程序会被暂停，没有任何响应，有点像卡死的感觉，这个停顿称为 STW。

代码清单 1-15 例子说明垃圾回收对应用程序产生的影响。模拟当生产环境系统出现严重性能问题，即尝试重现了 Stop-the-World 现象。

---

[1] 《灌篮高手》（SLAM DUNK）是日本漫画家井上雄彦以高中篮球为题材的励志型漫画及动画作品，是《周刊少年 Jump》全白金时代（20 世纪 90 年代上半叶）三大台柱漫画之一，也是日本历史上销量最高的漫画之一。20 世纪 90 年代，《灌篮高手》被引进中国，成为在中国影响最深远的动漫作品之一。

[2] 动漫作品《灌篮高手》的主人公。

[3] 赤木刚宪是日本动漫《灌篮高手》中的主要人物角色之一，原型为 NBA 球星帕特里克·尤因。

[4] 即 GC Event/Phase。

代码清单 1-15　Stop-the-World 示例

```
import Java.util.HashMap;

public class StopWorldDemo {
    public static class MyThread extends Thread{
     HashMap map = new HashMap();
     public void run(){
      try{
       while(true){
        if(map.size()*512/1024/1024>=400){
         map.clear();//防止内存溢出
         System.out.println("clean map");
        }
        byte[] b1;
        for(int i=0;i<100;i++){
         b1 = new byte[512];//模拟内存占用
         map.put(System.nanoTime(),b1);
        }
       }
      }catch(Exception ex){
       ex.printStackTrace();
      }
     }
    }

    public static class PrintThread extends Thread{
       public final long starttime = System.currentTimeMillis();

       public void run(){
        try{
         while(true){
          //每毫秒打印时间信息
          long t = System.currentTimeMillis()-starttime;
          System.out.println(t/1000+"."+t%1000);
          Thread.sleep(1000);
         }
        }catch(Exception ex){
         ex.printStackTrace();
        }
       }
    }
```

```
    public static void main(String[] args){
        MyThread t = new MyThread();
        PrintThread p = new PrintThread();
        t.start();
        p.start();
    }
}
```

上述代码中定义了两个线程，分别是 MyThread 和 PrintThread。MyThread 不停地申请系统内存，这会迫使进行垃圾回收，PrintThread 则每隔 0.1s 打印出系统启动时间。正常情况下应该 1s 有 10 个输出。

使用选项-Xmx512M -Xms512M -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails，输出如代码清单 1-16 所示。

代码清单 1-16　Stop-the-World 示例运行输出

```
0.0
1.14
clean map
2.29
3.309
clean map
4.324
clean map
5.807
clean map
6.978
clean map
7.993
clean map
9.8
clean map
10.148
11.522
clean map
12.755
clean map
```

GC 输出如代码清单 1-17 所示。

代码清单 1-17　Stop-the-World 示例的 GC 输出

```
   0.366: [GC 0.366: [DefNew: 139776K->17472K(157248K), 0.2675607 secs]
139776K->127626K(506816K), 0.2677332 secs] [Times: user=0.19 sys=0.08,
real=0.26 secs]
   0.788: [GC 0.788: [DefNew: 157248K->17471K(157248K), 0.3404208 secs]
267402K->261455K(506816K), 0.3405538 secs] [Times: user=0.31 sys=0.03,
real=0.34 secs]
   1.241: [GC 1.241: [DefNew: 157247K->157247K(157248K), 0.0000146 secs]1.241:
[Tenured: 243983K->349567K(349568K), 0.4988513 secs] 401231K->393834K(506816K),
[Perm : 380K->380K(12288K)], 0.4990866 secs] [Times: user=0.50 sys=0.00,
real=0.50 secs]
   1.940: [Full GC 1.940: [Tenured: 349567K->47656K(349568K), 0.1760066 secs]
506815K->47656K(506816K), [Perm : 380K->380K(12288K)], 0.1761751 secs] [Times:
user=0.17 sys=0.00, real=0.17 secs]
   2.226: [GC 2.227: [DefNew: 139776K->17472K(157248K), 0.2130968 secs]
187432K->185248K(506816K), 0.2132255 secs] [Times: user=0.22 sys=0.00,
real=0.22 secs]
   2.555: [GC 2.555: [DefNew: 157248K->17471K(157248K), 0.2630955 secs]
325024K->323914K(506816K), 0.2632084 secs] [Times: user=0.27 sys=0.00,
real=0.27 secs]
   2.922: [GC 2.922: [DefNew: 157247K->157247K(157248K), 0.0000138 secs]2.922:
[Tenured: 306443K->349567K(349568K), 0.5437268 secs] 463690K->455014K(506816K),
[Perm : 380K->380K(12288K)], 0.5439029 secs] [Times: user=0.53 sys=0.00,
real=0.53 secs]
   3.556: [Full GC 3.556: [Tenured: 349567K->51995K(349568K), 0.1789923 secs]
506815K->51995K(506816K), [Perm : 380K->379K(12288K)], 0.1791206 secs] [Times:
user=0.19 sys=0.00, real=0.19 secs]
   3.845: [GC 3.845: [DefNew: 139776K->17471K(157248K), 0.2132299 secs]
191771K->190545K(506816K), 0.2133538 secs] [Times: user=0.20 sys=0.00,
real=0.20 secs]
   4.174: [GC 4.174: [DefNew: 157247K->17471K(157248K), 0.2663958 secs]
330321K->329202K(506816K), 0.2665229 secs] [Times: user=0.27 sys=0.00,
real=0.26 secs]
   4.569: [GC 4.569: [DefNew: 157247K->157247K(157248K), 0.0000158 secs]4.569:
[Tenured: 311730K->12917K(349568K), 0.1204363 secs] 468978K->12917K(506816K),
[Perm : 379K->379K(12288K)], 0.1206203 secs] [Times: user=0.13 sys=0.00,
real=0.13 secs]
   4.798: [GC 4.798: [DefNew: 139776K->17471K(157248K), 0.2108548 secs]
152693K->151356K(506816K), 0.2109783 secs] [Times: user=0.20 sys=0.00,
real=0.20 secs]
   5.122: [GC 5.122: [DefNew: 157247K->17472K(157248K), 0.2431866 secs]
```

```
291132K->289949K(506816K), 0.2432983 secs] [Times: user=0.25 sys=0.00,
real=0.25 secs]
   5.467: [GC 5.467: [DefNew: 157248K->157248K(157248K), 0.0000134 secs]5.467:
[Tenured: 272477K->349567K(349568K), 0.4970060 secs] 429725K->420767K(506816K),
[Perm : 379K->379K(12288K)], 0.4971599 secs] [Times: user=0.48 sys=0.00,
real=0.48 secs]
   6.058: [Full GC 6.058: [Tenured: 349567K->41509K(349568K), 0.1328209 secs]
506815K->41509K(506816K), [Perm : 379K->379K(12288K)], 0.1329192 secs] [Times:
user=0.14 sys=0.00, real=0.14 secs]
   6.274: [GC 6.274: [DefNew: 139776K->17472K(157248K), 0.1368488 secs]
181285K->154007K(506816K), 0.1369451 secs] [Times: user=0.14 sys=0.00,
real=0.14 secs]
   6.495: [GC 6.495: [DefNew: 157248K->17471K(157248K), 0.1743854 secs]
293783K->268532K(506816K), 0.1744881 secs] [Times: user=0.19 sys=0.00,
real=0.19 secs]
   6.760: [GC 6.760: [DefNew: 157247K->157247K(157248K), 0.0000126 secs]6.760:
[Tenured: 251060K->349567K(349568K), 0.3610367 secs] 408308K->388050K(506816K),
[Perm : 379K->379K(12288K)], 0.3611760 secs] [Times: user=0.37 sys=0.00,
real=0.37 secs]
   7.227: [Full GC 7.227: [Tenured: 349567K->33158K(349568K), 0.1144929 secs]
506815K->33158K(506816K), [Perm : 379K->379K(12288K)], 0.1145849 secs] [Times:
user=0.13 sys=0.00, real=0.13 secs]
   7.414: [GC 7.414: [DefNew: 139776K->17471K(157248K), 0.1112127 secs]
172934K->132730K(506816K), 0.1113000 secs] [Times: user=0.11 sys=0.00,
real=0.11 secs]
   7.599: [GC 7.599: [DefNew: 157247K->17472K(157248K), 0.1426927 secs]
272506K->234470K(506816K), 0.1427827 secs] [Times: user=0.14 sys=0.00,
real=0.14 secs]
   7.823: [GC 7.823: [DefNew: 157248K->157248K(157248K), 0.0000107 secs]7.823:
[Tenured: 216998K->340621K(349568K), 0.3163073 secs] 374246K->340621K(506816K),
[Perm : 379K->379K(12288K)], 0.3164363 secs] [Times: user=0.31 sys=0.00,
real=0.31 secs]
   8.217: [GC 8.217: [DefNew: 139776K->139776K(157248K), 0.0000111 secs]8.217:
[Tenured: 340621K->349567K(349568K), 0.3631919 secs] 480397K->440326K(506816K),
[Perm : 379K->379K(12288K)], 0.3633182 secs] [Times: user=0.36 sys=0.00,
real=0.36 secs]
   8.643: [Full GC 8.643: [Tenured: 349567K->37244K(349568K), 0.1115731 secs]
506815K->37244K(506816K), [Perm : 379K->379K(12288K)], 0.1116607 secs] [Times:
user=0.11 sys=0.00, real=0.11 secs]
   8.821: [GC 8.821: [DefNew: 139776K->17472K(157248K), 0.0952736 secs]
177020K->127601K(506816K), 0.0953525 secs] [Times: user=0.11 sys=0.00,
```

```
real=0.11 secs]
   8.983: [GC 8.983: [DefNew: 157248K->17471K(157248K), 0.1197464 secs]
267377K->219190K(506816K), 0.1198281 secs] [Times: user=0.11 sys=0.00,
real=0.11 secs]
   9.171: [GC 9.171: [DefNew: 157247K->17472K(157248K), 0.1685964 secs]
358966K->312990K(506816K), 0.1686990 secs] [Times: user=0.17 sys=0.00,
real=0.17 secs]
   9.467: [GC 9.467: [DefNew: 157248K->157248K(157248K), 0.0000150 secs]9.467:
[Tenured: 295518K->349567K(349568K), 0.5542238 secs] 452766K->452375K(506816K),
[Perm : 379K->379K(12288K)], 0.5544066 secs] [Times: user=0.55 sys=0.00,
real=0.55 secs]
   10.113: [Full GC 10.114: [Tenured: 349567K->52002K(349568K), 0.1808479 secs]
506815K->52002K(506816K), [Perm : 379K->379K(12288K)], 0.1809750 secs] [Times:
user=0.19 sys=0.00, real=0.19 secs]
   10.407: [GC 10.407: [DefNew: 139776K->17472K(157248K), 0.2176954 secs]
191778K->190499K(506816K), 0.2178194 secs] [Times: user=0.22 sys=0.00,
real=0.22 secs]
   10.740: [GC 10.740: [DefNew: 157248K->17471K(157248K), 0.2684413 secs]
330275K->329159K(506816K), 0.2685582 secs] [Times: user=0.28 sys=0.00,
real=0.28 secs]
   11.122: [GC 11.122: [DefNew: 157247K->157247K(157248K), 0.0000134
secs]11.122: [Tenured: 311687K->349567K(349568K), 0.5415732 secs]
468935K->462097K(506816K), [Perm : 379K->379K(12288K)], 0.5417346 secs] [Times:
user=0.50 sys=0.00, real=0.55 secs]
   11.757: [Full GC 11.757: [Tenured: 349567K->48817K(349568K), 0.1491027 secs]
506815K->48817K(506816K), [Perm : 379K->379K(12288K)], 0.1492014 secs] [Times:
user=0.16 sys=0.00, real=0.16 secs]
   11.990: [GC 11.990: [DefNew: 139776K->17472K(157248K), 0.1398503 secs]
188593K->162017K(506816K), 0.1399383 secs] [Times: user=0.14 sys=0.00,
real=0.14 secs]
   12.253: [GC 12.253: [DefNew: 157248K->17471K(157248K), 0.2044428 secs]
301793K->293736K(506816K), 0.2045332 secs] [Times: user=0.20 sys=0.00,
real=0.20 secs]
   12.539: [GC 12.539: [DefNew: 157247K->157247K(157248K), 0.0000114
secs]12.539: [Tenured: 276264K->349567K(349568K), 0.3627016 secs]
433512K->403701K(506816K), [Perm : 379K->379K(12288K)], 0.3628280 secs] [Times:
user=0.36 sys=0.00, real=0.36 secs]
   12.986: [Full GC 12.986: [Tenured: 349567K->27685K(349568K), 0.1033825 secs]
506815K->27685K(506816K), [Perm : 379K->379K(12288K)], 0.1034725 secs] [Times:
user=0.11 sys=0.00, real=0.11 secs]
```

```
   13.154: [GC 13.154: [DefNew: 139776K->17472K(157248K), 0.0961716 secs]
167461K->116529K(506816K), 0.0962454 secs]
```

从上面的 GC 日志，可以看出从 1.14s 到 2.9s 之间出现了一次 Full GC，持续时间 0.17s，以及两次 Minor GC，这些 GC 严重干扰了 PrintThread 的正常工作。

当通过 Stop-the-World 机制的方式来运行垃圾收集器时，垃圾收集器会在内存回收的过程中暂停程序中所有的工作线程，直至完成内存回收才会恢复之前被暂停的工作线程。如果 Stop-the-World 出现在新生代的 Minor GC 中时，由于新生代的内存空间通常都比较小，所以暂停时间也在可接受的合理范围之内，不过一旦出现在老年代的 Full GC 中时，程序的工作线程被暂停的时间将会更久，这往往直接跟 Java 堆空间所管理的内存大小有关。简单来说，内存空间越大，执行 Full GC 的时间就会越久，相对的工作线程被暂停的时间也就会更长。以互联网项目为例，由于项目的特殊性，因此经常需要用到多达几十乃至上百 GB 的内存，如果用户的登录操作恰巧碰见垃圾收集器在执行 Full GC 时，这将会是一场灾难。如果用户等待的时间过长，这就不是用户体验下降这么简单的事情了，甚至有可能会流失用户，所以 JVM 的设计者们提供了并发回收希望以此缩短 Stop-the-world 机制的暂停时间。直到目前为止，哪怕是 G1 也不能完全避免 Stop-the-world 情况发生，只能说垃圾回收器越来越优秀，回收效率越来越高，尽可能地缩短了暂停时间。

STW 事件和采用哪款 GC 无关，所有的 GC 都有这个事件。被 STW 中断的应用程序线程会在完成 GC 之后恢复，频繁中断会让用户感觉像是网速不快造成电影卡带一样，所以我们需要减少 STW 的发生。另外，既然有“Stop-the-World”，那一定有能和 GC 事件/过程同时执行的 Java 应用线程。

注意，后续我都采用独占式来表示“Stop-the-World”，感觉这样更加贴近中国人的阅读感受。

1.2.2.15 对象存活判断

判断对象是否存活一般有两种方式。

- 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。
- 可达性分析（Reachability Analysis）：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的不可达对象。这个方法目前较为常用。

在 Java 语言中，GC Roots 包括以下内容。

- 虚拟机栈内引用的对象。
- 方法区中类静态属性实体引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈内 JNI 引用的对象。

1.2.2.16　Weighted Average

即加权平均值。将各数值乘以相应的权数，然后求和得到总体值，再除以总的单位数。平均数的大小不仅取决于总体中各单位的标志值（变量值）的大小，而且取决于各标志值出现的次数（频数），由于各标志值出现的次数对其在平均数中的影响起着权衡轻重的作用，因此叫作权数。

举两个示例。假设需要计算一个同学的某一科的考试成绩，平时测验 80，期中 90，期末 95。学校规定的科目成绩的计算方式，平时测验占 20%，期中成绩占 30%，期末成绩占 50%。

这里，每个成绩所占的比重叫作权重。那么，

加权平均值 =(80×20% + 90×30% + 95×50%)/(20%+30%+50%) =90.5

算术平均值(80 + 90 + 95)/3 = 88.3

上面的例子是已知权重的情况。下面的例子是未知权重的情况：

股票 A，1000 股，价格 10；

股票 B，2000 股，价格 15；

算术平均值 =(10 + 15) / 2 = 12.5；

加权平均值 =(10×1000 + 15×2000) / (1000 + 2000) = 13.33

其实，在每一个数的权数相同的情况下，加权平均值就等于算术平均值。

GC 在一些计算需求下会采用加权方式计算并得出结果。

1.2.2.17　阈值（Threshold）

中文翻译为阈值，这个概念被广泛用于计算机程序设计，特别是 JVM 内部。阈值在 JVM 内部通常是指某一个界限，一旦超过或者低于这个界限，JVM 进程会触发一个或者一系列动作，这些动作通常是与垃圾回收相关的。

1.2.2.18 Twitter 自研 VM 的研究

第十届国际软件开发者大会在旧金山召开[1]，11 月 7 日首日会上，来自 Twitter JVM 组的 John Coomes 先生[2]介绍了团队基于 OpenJDK 的一个分支设立了自己的小组，并介绍了在此基础上所做的开发、上线、维护公司内部 JDK 版本的具体情况。

Twitter JDK 一般情况下每个月推出一个版本，但是也有个别紧急的情况会有短周期的开发上线情况，目前该 JDK 支持 OS X[3]和 Linux。Coomes 介绍 Twitter 试图保持与 OpenJDK 之间有的放矢的差异点的情况，和 Oracle 公司针对它们的 JDK 与 OpenJDK 之间的情况是比较相似的。现场有人提问 Coomes 关于合并 OpenJDK 最新上线功能和代码的情况，Coomes 介绍 Twitter 的态度是首先分析上线的功能是否为 Twitter 需要，即不盲目地跟随，不会为了维护和 OpenJDK 基线版本的紧耦合关系而毫无目的地合并代码。

Coomes 介绍了 Twitter JDK 的新功能包括扩展堆分析、异步 GC 日志、一个叫作 dubbed Contrail 的二进制日志框架、针对 G1 收集器的中间年代设计、额外的接口以及命令行选项、性能提升以及缺陷修复。Coomes 承诺一旦时机成熟，Twitter 会开源这些修改代码。

Coomes 介绍了一些 JVM 现存的实际问题，特别提到了长时间的 GC 停顿（每一款 GC 都有这个问题，虽然都有所改进，但是一直没有完全解决）以及在 OpenJDK 上面如何管理长时间停顿的一些办法。一个方法是隔离输入/输出强度、延迟加载比较敏感的作业。这个听起来好像没什么意思，但是你要想想有上千个 Twitter 的服务使用 Scala 和 Java，两者都需要运行在 JVM 上面，JVM 的输入/输出操作可以在安全点阶段被关闭，但是这样会造成对于 JVM 监控的能力下降。Coomes 提到了基于 Python 的服务对于基于 JVM 的语言没有太大的帮助，推测 Twitter 内部基于 JVM 的服务数量只会不断上涨。Twitter JDK 通过部署异步 GC 日志和 jvmstat 工具，处理了一些棘手的问题，这样的结果导致 GC 暂停阶段不会阻塞 I/O。Commes 还提到了未来 Contrail 工具（一个异步 GC 日志工具，它基于 JIT 编译器，实时运行并且具有堆栈跟踪能力）的上线节奏。Twitter JVM 优化措施还包括实施了贝叶斯优化器，这是一个从机器学习领域借鉴过来的技术，这样 JVM 可以迭代地、高效地学习成本函数噪音，导致更快地发现优化处理路径。

关于什么参数有利于 Twitter JDK 对应的 VM，Coomes 提到在 TwitterJVM 内部大约有 30 个不停的参数，需要我们科学地去选择解决优化。被用来测算 JVM 性能的成本函数是一个每秒

---

[1] 2016 年 11 月 7 日—11 月 11 日。

[2] 团队成员，两个孩子的父亲，11 月 8 日 Twitter 发文说参加大会同时也拿到了旧金山地区的超速罚单。

[3] OS X 是苹果公司为 Mac 系列产品开发的专属操作系统，OS X 是全世界第一个基于 FreeBSD 系统采用“面向对象操作系统”的全面的操作系统。

请求数量，这个每秒请求数量基于耗费在 GC 上的时钟时间。Twitter 在 Staging 环境上让自己新的 VM 在全部 30 个参数运行情况下循环地运行了 70 次，生成的报告显示性能提升了 182%。未来的工作需要高强度的测试或者产品红线控制，更长时间的实验周期，基于 Mesos 的并发实现，并且希望能够具有在较早阶段自动检测和终止差性能实现的能力。

1.2.2.19　System.gc()方法

在默认情况下，System.gc()会显式直接触发 Full GC，同时对老年代和新生代进行回收。而一般情况下，垃圾回收应该是自动进行的，无须手工触发，否则就太过于麻烦了。如果过于频繁地触发垃圾回收，对于系统的整体性能是没有好处的。因此虚拟机提供了一个选项 DisableExplicitGC 来控制是否手工触发 GC。

使用 System.gc()可以不管 JVM 使用的是哪一种垃圾回收的算法，都可以显式地请求 Java 的垃圾回收。可以看一个程序示例，如代码清单 1-18 所示。

代码清单 1-18　System.gc 示例

```
class TestGC
{
    public static void main(String[] args)
    {
        new TestGC();
        System.gc();
        System.runFinalization();
     }
}
```

在这个例子中，一个新的对象被创建。由于它没有被使用，所以该对象迅速地变为不可达（Unreachable），程序编译后执行命令 Java -verbosegc TestG，对应运行结果为[Full GC 168K->97K(1984K)， 0.0253873 secs]。箭头前后的数据 168K 和 97K 分别表示垃圾收集 GC 前后所有存活对象使用的内存容量，说明有 168K-97K=71K 的对象容量被回收，括号内的数据 1984K 为堆内存的总容量，收集所需要的时间是 0.0253873 秒（这个时间在每次执行的时候会有所不同）。

Runtime.gc()是一个 native 方法，最终实现在 jvm.cpp 中，如代码清单 1-19 所示。

代码清单 1-19　Runtime.gc()源代码

```
JVM_ENTRY_NO_ENV(void,JVM_GC(void))
  JVMWrapper("JVM_GC");
```

```
    If(!DisableExplicitGC){
       Universe:heap()->collect(GCCause::_Java_lang_system_gc);
    }
JVM_END
```

默认情况下，即使 System.gc()生效，它会使用传统的 Full GC 方式回收整个堆内存垃圾，而会忽略选项中的 UseG1GC，比如我们使用如下选项-XX:+PrintGCDetails –XX:+UseG1GC 运行代码清单 1-18 程序，运行结果输出如代码清单 1-20 所示。

代码清单 1-20　强制使用 System.gc 后代码清单 1-18 运行输出

```
[Full GC (System.gc())  908K->526K(8192K), 0.0250794 secs]
   [Eden: 1024.0K(7168.0K)->0.0B(4096.0K) Survivors: 0.0B->0.0B Heap:
908.8K(60.0M)->526.7K(8192.0K)], [Metaspace: 2551K->2551K(1056768K)]
 [Times: user=0.00 sys=0.00, real=0.02 secs]
Heap
 garbage-first heap   total 8192K, used 526K [0x00000000c4200000,
0x00000000c4300040, 0x0000000100000000)
  region size 1024K, 1 young (1024K), 0 survivors (0K)
 Metaspace       used 2561K, capacity 4486K, committed 4864K, reserved
1056768K
  class space    used 286K, capacity 386K, committed 512K, reserved 1048576K
```

从代码清单 1-20 可以看出，G1 GC 并没有并发执行，因为在日志中没有任何并发相关信息。打开虚拟机选项-XX:+ExplicitGCInvokesConcurrent 后，可以改变这种默认的行为，输出如代码清单 1-21 所示。

代码清单 1-21　开启选项–XX:+ExplicitGCInvokesConcurrent 后代码清单 1-18 运行输出

```
[GC pause (System.gc()) (young) (initial-mark), 0.0018781 secs]
   [Parallel Time: 1.6 ms, GC Workers: 2]
      [GC Worker Start (ms): Min: 397.4, Avg: 397.5, Max: 397.5, Diff: 0.0]
      [Ext Root Scanning (ms): Min: 0.5, Avg: 0.5, Max: 0.5, Diff: 0.0, Sum: 0.9]
      [Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0]
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Object Copy (ms): Min: 1.0, Avg: 1.0, Max: 1.1, Diff: 0.0, Sum: 2.1]
      [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Termination Attempts: Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 2]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.0, Sum: 0.1]
      [GC Worker Total (ms): Min: 1.6, Avg: 1.6, Max: 1.6, Diff: 0.0, Sum: 3.2]
```

```
        [GC Worker End (ms): Min: 399.0, Avg: 399.0, Max: 399.0, Diff: 0.0]
      [Code Root Fixup: 0.0 ms]
      [Code Root Purge: 0.0 ms]
      [Clear CT: 0.0 ms]
      [Other: 0.2 ms]
        [Choose CSet: 0.0 ms]
        [Ref Proc: 0.1 ms]
        [Ref Enq: 0.0 ms]
        [Redirty Cards: 0.0 ms]
        [Humongous Register: 0.0 ms]
        [Humongous Reclaim: 0.0 ms]
        [Free CSet: 0.0 ms]
      [Eden: 1024.0K(7168.0K)->0.0B(5120.0K) Survivors: 0.0B->1024.0K Heap:
908.8K(60.0M)->608.1K(60.0M)]
    [Times: user=0.02 sys=0.00, real=0.00 secs]
   [GC concurrent-root-region-scan-start]
   [GC concurrent-root-region-scan-end, 0.0006240 secs]
   [GC concurrent-mark-start]
   [GC concurrent-mark-end, 0.0001534 secs]
   [GC remark [Finalize Marking, 0.0000274 secs] [GC ref-proc, 0.0000553 secs]
[Unloading, 0.0016140 secs], 0.0018180 secs]
    [Times: user=0.00 sys=0.00, real=0.00 secs]
   [GC cleanup 608K->608K(60M), 0.0002289 secs]
    [Times: user=0.00 sys=0.00, real=0.00 secs]
   Heap
    garbage-first heap   total 61440K, used 608K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 2 young (2048K), 1 survivors (1024K)
    Metaspace       used 2561K, capacity 4486K, committed 4864K, reserved
1056768K
     class space    used 286K, capacity 386K, committed 512K, reserved 1048576K
```

请注意，调用 System.gc()也仅仅是一个请求垃圾回收建议。JVM 接受这个消息后并不是立即做垃圾回收，而只是对几个垃圾回收算法做了加权，使垃圾回收操作容易发生，或提早发生，或回收较多而已。所以，JVM 是有自己思维的，不会乱用洪荒之力。

1.2.2.20　堆外内存（off-heap Memory）

JVM 内部会把所有内存分成 Java 使用的堆内存和 Native 使用的内存，它们之间是不能共享的，就是说当你的 Native 内存用完了时，如果 Java 堆又有空闲的内存，这时 Native 会重新向 JVM 申请，而不是直接使用 Java 堆内存。

线程栈、应用程序代码、NIO 缓存用的都是堆外内存。事实上在 C 或者 C++中，你只能使用未托管内存，因为它们默认是没有托管堆的。在 Java 中使用托管内存或者堆内存是这门语言比较独特的一个特性[1]。

使用堆外内存与对象池[2]都能减少 GC 的暂停时间，这是它们唯一的共同点。生命周期短的可变对象，创建开销大，或者生命周期虽然长，但是存在冗余的可变对象，这两类都比较适合使用对象池。生命周期适中，或者复杂的对象则比较适合由 GC 来进行处理。然而，中长生命周期的可变对象也适用于堆外内存。

堆外内存的好处包括如下 4 点。

- 可以扩展至更大的内存空间，比如超过 1TB 甚至比主内存还大的空间。
- 理论上可以减少 GC 暂停时间。
- 可以在进程间共享，减少 JVM 间的对象复制，使得 JVM 的分割部署更容易实现。
- 持久化存储可以支持快速重启，同时还能够在测试环境中重现生产数据。

高性能计算领域最大的一个难点在于重现那些隐蔽的 BUG，并证实问题已经得到修复。通过将输入事件及持久化的形式存储到堆外内存中，你可以将你的关键系统变成一系列的复杂状态机。

使用堆外内存会引入的问题如下所述。

- 数据结构变得有些别扭。
- 可能需要一个简单的数据结构以便于直接映射到堆外内存。
- 使用复杂的数据结构并序列化及反序列化到内存中。
- 序列化比使用堆对象的性能差很多。

与其他实现可伸缩性的方案相比，比如堆缓存、消息队列，使用对外内存要更为简单、高效。堆外内存可以支持操作系统的同步写入，不再需要异步去执行，但是可能会丢失数据。对外内存的好处是能够缩短映射时间，映射 1TB 的数据只需要 10ms。

1.2.2.21 finalize()方法对垃圾回收的影响

Java 中提供了一个类似 C++中析构函数的机制，即 finalize()函数。finalize()函数允许在子

---

[1] 不过 Java 并不是唯一这么做的语言。

[2] Java5 之前，几乎所有对象，包括对象池本身，都通过对象池来提升性能，而之后由于创建对象的开销大幅下降，只有线程、Socket、数据库连接才继续保持对象池的使用。

类中被重载，用于在对象被回收时进行资源释放。一般不建议采用该方法进行资源释放，理由包括下面三点。

- 在 finalize()时可能会导致对象复活。
- finalize()函数的执行时间是没有保障的，它完全由 GC 线程决定，极端情况下，若不发生 GC，则 finalize()方法将没有执行机会。
- 一个糟糕的 finalize()会严重影响 GC 的性能。

函数 finalize()是由 FinalizerThread 线程处理的。每一个即将被回收的并且包含有 finalize()方法的对象都会在被正式回收前加入 FinalizerThread 的执行队列，该队列内部由链表结构实现，队列中每一项为 Java.lang.ref.Finalizer 引用对象，本质为一个引用。由于对象在回收前被 Finalizer 的 referent 字段进行强引用操作，并加入了 FinalizerThread 的执行队列，这意味着对象又称为可达对象，不会再被正常回收。由于在引用队列中的元素排队执行 finalize()方法，一旦出现性能问题，将导致这些垃圾对象长时间堆积在内存中，可能会导致 OOM 异常。

不过 finalize()方法也有它起作用的时候。比如在 MySQL 的 JDBC 驱动类 ConnectionImpl 中就实现了 finalize()方法。当一个 JDBC Connection 被回收时，需要进行连接的关闭，即调用 cleanup()方法。事实上，在回收前，开发人员如果正常调用了 Connection.close()方法，那么连接就会被显式关闭，那样的话，在 cleanup()方法中将什么都不会做。而如果忘记显式关闭连接，而 Connection 对象又被回收了，则会隐式地进行连接的关闭，确保没有数据库连接的泄漏。finalize()方法在这里作为一种双保险措施，在正常方法出现意外时进行补位，尽可能确保系统稳定。当然，由于其调用时间的不确定性，这个操作不能单独作为可靠的资源回收手段。

1.2.2.22　markOop

markOop 描述了一个对象(也包括了 Class)的状态信息，Java 语法层面的每个对象或者 Class 在 JVM 的结构表示中都会包含一个 markOop 作为 Header，当然还有一些其他的 JVM 数据结构也用它做 Header。markOop 由 32 位或者 64 位构成，具体位数根据运行环境而定。

markOop 的值根据所描述的对象的类型（比如是锁对象还是正常的对象）以及作用的不同而不同。就算在同一个对象里，它的值也是可能会不断变化的，比如锁对象，在一开始创建的时候其实并不知道是锁对象，会当成一个正常对象来创建（在对象的类型并没有设置偏向锁的情况下，其 markOop 值可能是 0x1），但是随着执行到 synchronized 的代码逻辑时，就知道其实它是一个锁对象了，它的值就不再是 0x1 了，而是一个新的值，该值是对应栈帧结构里的监控对象列表里的某一个内存地址。

1.2.2.23 JVM 虚拟化前景

传统的 Java 应用部署模式，一般遵循“硬件→操作系统→JVM→Java 应用”这种自底向上的部署结构，其中 Java EE 应用可以细化为“硬件→操作系统→JVM→JavaEE 容器→JavaEE 应用”的部署结构。这种部署结构往往比较重，操作系统、JVM 和 JavaEE 容器造成 overhead 很高，而很多时候一个 Java 应用并不需要跑满整个硬件的资源，导致这种模式的资源利用率是比较低的。

而另一方面，硬件虚拟化技术逐渐成熟：VMware Hypervisor、Xen、KVM、Power LPAR 等技术能够帮助我们在同一个硬件上部署多个操作系统实例（每个操作系统实例可以理解为宿主机的租户），而时下流行的 OS Container 技术如 LXC、Docker 等，则是把操作系统虚拟化为多个实例，实现更轻量级的虚拟化。无论哪个层面的虚拟化，其目的都是对资源利用率更加高效的追求。

同样的思路，也可以在 JVM 层面，或者容器框架层面做虚拟化，类似于 Hypervisor 或者 OS Container，让虚拟化的 JVM/容器框架可以支持多租户的运行模式，这是比 OS 虚拟化更高一层的做法。随着虚拟化层次的提高，从 Hardware 到 OS，到 JVM，再到容器，单个租户应用的部署成本也在下降，最右边的“多租户”的 JavaEE Container（Multitenant-Container）是在容器框架层面的虚拟化，可以支持更高密度的应用部署，而不是传统的 APP : Container = 1:1 的部署方式。

什么是单个租户应用部署成本？举一个简单的例子，在没有虚拟化之前，传统的部署模式，一个 JavaEE 应用需要独占所有的硬件资源（CPU、MEM、网络等）。Hypervisor 的出现，使得一个共享的硬件资源上可以同时跑多个 OS，这种资源使用上的节约本质上是通过 Over-Commit（即允许租户超量使用物理资源）来达到的，即假设跑在同一个虚拟化环境的不同租户不会在同一个时间消费同样的资源。同样的道理可以来理解 JVM/容器框架的虚拟化。

多个 JavaEE 应用可以部署在同一个 JVM/容器内，但逻辑上它们各自会认为是运行在一个独立的 JVM/容器内，从而可以更大程度地提高资源利用率，降低单个租户应用的部署成本。

1.2.3 G1 涉及术语

1.2.3.1 Metaspace

JDK8 HotSpot JVM 使用本地内存来存储类元数据信息并称为元空间（Metaspace），这与

Oracle JRockit 和 IBM JVM 很相似。总的来说这是一个好消息，因为这么一来就意味着不会再有 Java.lang.OutOfMemoryError: PermGen 问题出现了[1]，也不再需要你进行针对该区域的调优及监控内存空间的使用，即如果在启用时设置了 PermSize 和 MaxPermSize 这两个选项，选项本身的设置会被忽略，并且 JVM 会给出警告。

默认情况下，大部分类元数据都在本地内存中分配，类元数据只受可用的本地内存限制（容量取决于是 32 位或是 64 位操作系统的可用虚拟内存大小）。新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

一般情况下，适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收情况如果频繁发生，说明可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

举一个例子，假设程序在 JDK7 和 JDK8 上分别运行，并且程序会需要加载大量的类，那么在 JDK7 版本，当加载到几万个类时，PermGen 被耗尽，会抛出错误 Java.lang.OutOfMemoryError:PermGen。而使用 JDK8 时，由于 JVM Metaspace 进行了动态扩展，本地内存的使用会由几十 MB 增长到几百 MB，以满足程序中不断增长的类数据内存占用需求。可以观察到 JVM 的垃圾回收事件试图销毁僵死的类或类加载器对象。但是，由于程序的泄漏，JVM 别无选择只能动态扩展 Metaspace 内存空间。程序能够运行几万次迭代，加载超过几万个类，而没有出现 OOM 事件，这都是 Metaspace 的功劳。

G1 GC 与 Metaspace 相关的选项如下。

- -XX:MetaspaceSize：初始化元空间的大小（默认 12Mbytes 在 32bit client VM and 16Mbytes 在 32bit server VM，在 64bit VM 上会更大些）。
- -XX:MaxMetaspaceSize：最大元空间的大小（默认本地内存）。
- -XX:MinMetaspaceFreeRatio：扩大空间的最小比率，当 GC 后，内存占用超过这一比率，就会扩大空间。
- -XX:MaxMetaspaceFreeRatio：缩小空间的最小比率，当 GC 后，内存占用低于这一比率，就会缩小空间。

[1] 从 JDK8 开始移除了永久区概念，转而使用元空间。

1.2.3.2 Mixed GC Event

即混合 GC 事件，在这个事件内部，所有的年轻代 Region 和一部分老年代 Region 一起被回收。混合 GC 事件一定是跟在 Minor GC 之后的，这个第 4 章会详细介绍，并且混合 GC 只有在存活对象元数据存在的情况下才会触发。

来看一段混合 GC 的日志输出，如代码清单 1-22 所示。

代码清单 1-22　混合 GC 日志范例

```
   [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions
available, candidate old regions: 363 regions, reclaimable: 9830652576 bytes
(10.40 %), threshold: 10.00 %]
   [Parallel Time: 145.1 ms, GC Workers: 23]
   [GC Worker Start (ms): Min: 251176.0, Avg: 251176.4, Max: 251176.7, Diff:
0.7]
   [Ext Root Scanning (ms): Min: 0.8, Avg: 1.2, Max: 1.7, Diff: 0.9, Sum: 28.1]
   [Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.6, Sum: 5.8]
   [Processed Buffers: Min: 0, Avg: 1.6, Max: 9, Diff: 9, Sum: 37]
   [Scan RS (ms): Min: 6.0, Avg: 6.2, Max: 6.3, Diff: 0.3, Sum: 143.0]
   [Object Copy (ms): Min: 136.2, Avg: 136.3, Max: 136.4, Diff: 0.3, Sum: 3133.9]
   [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
   [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 1.9]
   [GC Worker Total (ms): Min: 143.7, Avg: 144.0, Max: 144.5, Diff: 0.8, Sum:
3313.0]
   [GC Worker End (ms): Min: 251320.4, Avg: 251320.5, Max: 251320.6, Diff: 0.2]
   [Code Root Fixup: 0.0 ms]
   [Clear CT: 6.6 ms]
   [Other: 26.8 ms]
   [Choose CSet: 0.2 ms]
   [Ref Proc: 16.6 ms]
   [Ref Enq: 0.9 ms]
   [Free CSet: 2.0 ms]
   [Eden: 3904.0M(3904.0M)->0.0B(4448.0M) Survivors: 576.0M->32.0M Heap:
63.7G(88.0G)->58.3G(88.0G)]
   [Times: user=3.43 sys=0.01, real=0.18 secs]
```

从这段混合 GC 事件发生之后输出的日志来看，G1 GC 日志的层次是非常清晰的。日志列出了这次暂停发生的时间、原因，并分级各种线程所消耗的时长以及 CPU 时间的均值、最大值和最小值。最后，G1 GC 列出了本次暂停的清理结果，以及总共消耗的时间。

1.2.3.3　Reclaimable

G1GC 为了能够回收，创建了一系列专门用于存放可回收对象的 Region。这些 Region 都在一个链表队列里面，这个队列只包含存活率小于-XX:G1MixedGCLiveThresholdPercent（默认 85%）的 Region。Region 的值除以整个 Java 堆区，如果大于-XX:G1HeapWastePercent（默认 5%），则启动回收机制。

1.2.3.4　RSet

全称 Remembered Set，简称 RSet，即跟踪指向某个堆区（Region）内的对象引用。

在标记存活对象时，G1 使用 RememberSet 的概念，将每个分区外指向分区内的引用记录在该分区的 RememberSet 中，避免了对整个 Heap 的扫描，使得各个分区的 GC 更加独立。堆内存中的每个区都有一个 RSet，RSet 的作用是让堆区能并行独立地进行垃圾集合。RSet 所占用的 JVM 内存小于总大小的 5%。

在这样的背景下，可以看出 G1 GC 大大提高了触发 Full GC 时的 Heap 占用率，同时也使得 Minor GC 的暂停时间更加可控，对于内存较大的环境非常友好。

G1 GC 引入了一些新的选项。G1RSetUpdatingPauseTimePercent 设置 STW 阶段（独占阶段）为 G1 收集器指定更新 RememberSet 的时间占总 STW 时间的期望比例，默认为 10。而 G1ConcRefinementThreads 则是在程序运行时维护 RememberSet 的线程数目。通过对这两个值的对应调整，我们可以把 STW 阶段的 RememberSet 更新工作压力更多地移到并行阶段。

1.2.3.5　CSet

全称 Collection Set，简称 CSet，即收集集合，保存一次 GC 中将执行垃圾回收的区间（Region）。GC 时在 CSet 中的所有存活数据（Live Data）都会被转移（复制/移动）。集合中的堆区可以是 Eden，Survivor 和/或 Old Generation。CSets 所占用的 JVM 内存小于总大小的 1%。

从这里可以知道，实际上 CSet 相当于一个大圈，里面包含了很多的小圈（RSet），这些圈圈都是需要被回收的信息。这样可以把 CSet 比作垃圾场，RSet 是垃圾场里面一个个绿色的可回收垃圾桶[1]。

---

[1] 倡导一下垃圾分类，按照不同的垃圾性质放入不同颜色的垃圾桶，这个制度在杭州已经实行了 6 年，从 2010 年开始，希望全国推广，环境越来越好，这也是中国梦的一部分。

1.2.3.6 G1 Pause Time Target

从字面上理解就是 G1 停顿目标时间，由于垃圾收集阶段可能是独占式的，所以就会引起应用程序停顿，这个停顿时间就是你所设置的期望时间。G1 使用了一个停顿预测模型去匹配用户设定的目标停顿时间，并且基于这个目标停顿时间去选择需要回收的 Region 的数量。

1.2.3.7 Root Region Scan

这个阶段从根区间的扫描开始，标记所有可达的存活对象。由于在并行标记的执行过程中移动数据会造成应用程序暂停，所以根区间扫描这个阶段需要在下一次评估中断开始执行直到结束。

1.2.3.8 PLAB

全称为 Promotion Local Allocation Buffers，它被用于年轻代回收。PLAB 的作用是避免多线程竞争相同的数据，处理方式是每个线程拥有独立的 PLAB，用于针对幸存者和老年空间。当应用开启的线程较多时，最好使用-XX:-ResizePLAB 来关闭 PLAB()的大小调整，以避免大量的线程通信所导致的性能下降。

1.2.3.9 TLAB

全称为 Thread Local Allocation Buffers，即线程本地分配缓存，是一个线程专用的内存分配区域。

总的来说，TLAB 是为了加速对象分配而生的。由于对象一般会分配在堆上，而堆是全局共享的。因此在同一时间，可能会有多个线程在堆上申请空间。因此，每一次对象分配都必须要进行同步，而在竞争激烈的场合分配的效率又会进一步下降。考虑到对象分配几乎是 Java 最最常用的操作，所以 JVM 就使用了 TLAB 这种线程专属的区间来避免多线程冲突，提高对象分配的效率。TLAB 本身占用了 Eden 区的空间，即 JVM 会为每一个 Java 线程分配一块 TLAB 空间。

对于 G1 GC 来说，TLAB 是 Eden 的一个 Region，被一个单一线程用于分配资源。主要用途是让一个线程通过栈操作方式独享内存空间，用于对象分配，这样比多个线程之间共享资源要快很多。如果每个线程的分配内存不够，那么它会去全局内存池申请新的内存。这样也就是说，如果 TLAB 值设置过小，容易造成频繁申请，也就会造成 GC 性能下降。反之，如果设置过大，会造成 TLAB 使用不完，也就是说内存浪费。

1.2.3.10　Lock-free Manner[1]

这是针对第 4 章会涉及的 TLAB 调用本地线程的无锁竞争分配方式。

对于 GC 来说，堆内存的压缩和栈检索通常是引起性能问题的瓶颈爆发点。基于锁机制的算法横向扩展能力不太好，很容易达到瓶颈，所以采用基于 CAS/MCAS 同步方式可以确保不会被其他线程阻塞。

1.2.3.11　Region

G1 收集器将堆进行分区，划分为一个个的区域，每次收集的时候，只收集其中几个区域，以此来控制垃圾回收产生的停顿时间。这个区域就是这里介绍的单词：Region。

从字面上来说，Region 表示一个区域，每个区域里面的字母代表不同的分代内存空间类型（如[E]Eden,[O]Old,[S]Survivor），空白的区块不属于任何一个分区。G1 可以在需要的时候任意指定这个区域属于 Eden 或是 O 区之类的。

G1 通过将内存空间分成区域（Region）的方式避免内存碎片问题，每个 Region 是大小一致的，从逻辑层面来说，它们又是连续的虚拟内存块。G1 的并行全局标记阶段会决定整个堆区的存活对象，在这个标记阶段完成之后，G1 就知道哪些 Region 是空的了。

1.2.3.12　Ergonomics Heuristic Decision

在很多英文书里都能看到这串单词，特别是 Ergonomics Heuristic，它们的字面意思是人体工程学，可以理解为适合人类理解的行为、习惯。你会在第 3 章的 GC 日志里面看到 Ergonomics 这个单词，它后面一般跟着的是 G1 GC 相关的详细描述，比如堆内存日志、CSet 划分等，通常采用选项-XX:+PrintAdaptiveSizePolicy 时会看到这个单词。

1.2.3.13　Evacuation Failure

也可以理解为提升失败（Promotion Failure）、to-space exhaustion、to-space overflow。这个异常通常发生在提升对象时发现内存空间不足。对于这个异常，一般的做法是立即扩展堆内存，但是堆内存总有一个最大值，所以 GC 会让一些已经拷贝成功的引用进入老年代。对于 G1 GC，拷贝不成功的对象会被立即放入老年代。

---

[1] 推荐看一篇论文，*A Study of Lock-Free Based Concurrent Garbage Collectors for Multicore Platform*。

1.2.3.14 Top-at-mark-start

每个区间记录着两个 TAMS 指针（Top-at-mark-start），分别为 prevTAMS 和 nextTAMS。在 TAMS 以上的对象是新分配的，因而被视为隐式标记。

1.3 本章小结

本章节我们首先回顾了 JDK 的发展过程，然后开始具体讲解 Java 通用术语、GC 通用术语，以及 G1 GC 的独有术语。通过这一个章节的准备工作，可以进入后续章节的学习，特别是第 3 章、第 4 章，需要提前了解 G1 GC 的相关专业术语，才能深入了解、深入分析 GC 日志。

2 第2章 JVM & GC 深入知识

无论你是否承认，韩国电影目前代表了亚洲最高水平。奉俊昊[1]执导的电影《雪国列车》，改编自获得 1986 年昂格莱姆[2]国际漫画节大奖的法国同名科幻漫画原著。故事讲述一场突如其来的气候异变让地球上大部分人类灭亡，在一列没有终点、沿着铁轨一直行驶下去的列车上，载着地球上最后幸存的人们，“雪国列车”成为了他们最后的归宿、最后的信仰和最后的牢笼，在这里，受尽压迫的末节车厢反抗者为了生存与尊严向列车上的权利阶层展开斗争。这一节节的车厢，对应的是整个垃圾回收过程，残酷而现实。

垃圾回收是一个跟踪过程，它传递性地跟踪指向当前使用的对象的所有指针，以便找到可以引用的所有对象，然后重新使用在此跟踪过程中未找到的任何堆内存。公共语言运行库垃圾回收器还压缩使用中的内存，以缩小堆所需要的工作空间。这是对于垃圾回收机制的传统定义。

---

[1] 奉俊昊（ Bong Joon ho），1969 年 9 月 14 日出生于韩国大邱广域市，毕业于延世大学社会系（88 级），导演、编剧。

[2] 欧洲最大、资格最老的连环画艺术节，昂格莱姆国际连环画艺术节每年都会为本年度漫画风云人物颁发特别贡献奖，以此表彰其对漫画事业做出的杰出贡献。

垃圾回收是 Java 应用程序可持续运行的基础保障，它对于每个对象按照分代的方式进行切分，类同于雪国列车的各节车厢，每节车厢所搭乘的乘客不同，也会在一定时间被回收、清空，对象的生命周期会按照一定的规则进行设定。只有了解 JVM[1]、了解 GC[2]，才能真正编写高效的 Java 应用程序。

本章主要介绍和解决以下问题，这些也是全书的基础。

- 了解 JVM 和内存相关的知识。
- 了解常见的垃圾收集算法，这是 GC 实现的根本目标。
- 了解 GC 的基本概念。
- 了解各类 GC 的特性。
- 为深入了解 G1 GC 做好知识储备。

2.1 Java 虚拟机内存模型

Java 虚拟机内存模型是 Java 程序运行的基础，毕竟所有的程序都需要使用内存，这就需要 Java 虚拟机专门设计内存使用方式。Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干不同的数据区域，这些区域都有各自的用途以及创建和销毁的时间。

Java 虚拟机所管理的内存由几个运行时的数据区域组成，为了能使 Java 应用程序正常运行，Java 虚拟机将其内存数据分为程序计数器、虚拟机栈、本地方法栈、Java 堆、方法区等五个部分。简单的一句话描述用途，如图 2-1 所示，程序计数器用于存放下一条运行的指令，虚拟机栈和本地方法栈用于存放函数调用堆栈信息，Java 堆用于存放 Java 程序运行时所需的对象等数据，方法区用于存放程序的类元数据信息。其中，Java 堆和 Java 栈这两个内存区域是大多数 Java 程序员最关注的，也是最容易出现程序异常的区域，两个英文单词连起来读，听着像 Hip-Hop[3] 的感觉。

---

[1] 即 Java Virtual Machine。

[2] 即 Garbage Collection。

[3] Hip Hop 是起源于源自于街头的一种黑人文化，也泛指 rap（说唱乐）。

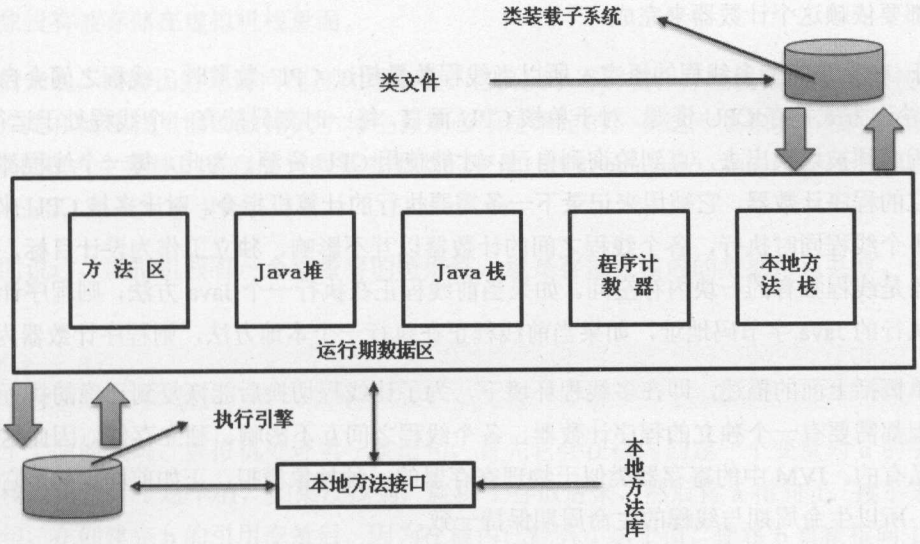


图 2-1　Java 虚拟机内存模型

如果根据受访权限的不同设置，可以定义上述几个区域分为线程共享和线程私有两大类。线程共享指的是可以允许被所有的线程共享访问的一类内存区域，这类区域包括堆内存区、方法区、运行时常量池等三个内存区域。

2.1.1　程序计数器

冯·诺伊曼计算机体系结构的主要内容之一就是“程序预存储，计算机自动执行”，处理器要执行的程序（指令序列）都是以二进制代码序列方式预存储在计算机的存储器中，处理器将这些代码逐条地抽取到处理器中再译码、执行，用以完成整个程序的执行。为了保证程序能够连续地执行下去，CPU 必须具有某些手段来确定下一条取指指令的地址，程序计数器正是起到了这种作用，所以通常又被称为“指令计数器”。

程序计数器，英文全称 Program Counter Register，它是一块很小的内存空间，它是运行速度最快的存储区域，这是因为它位于不同于其他存储区的地方——处理器内部。寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。实际上在 Java 应用程序内部不能直接控制寄存器，也不能在程序中感觉到寄存器存在的任何迹象。可以把程序计数器看作当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器的工作就是通过改变程序计

数器的值来选择下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都要依赖这个计数器来完成。

由于 Java 是支持多线程的语言，所以当线程数量超过 CPU 数量时，线程之间会自动根据时间片[1]轮询方式抢夺 CPU 资源。对于单核 CPU 而言，每一时刻只能有一个线程处于运行状态，其他线程必须被切换出去，直到轮询到自己，才能使用 CPU 资源。为此，每一个线程都必须有一个独立的程序计数器，它被用来记录下一条需要执行的计算机指令。对于多核 CPU 来说，可以允许多个线程同时执行，各个线程之间的计数器以互不影响、独立工作为设计目标，所以程序计数器是线程独有的一块内存空间。如果当前线程正在执行一个 Java 方法，则程序计数器记录正在执行的 Java 字节码地址，如果当前线程正在执行一个本地方法，则程序计数器为空。

简单概括上面的描述，即在多线程环境下，为了让线程切换后能恢复到正确的执行位置，每个线程都需要有一个独立的程序计数器，各个线程之间互不影响、独立存储，因此这块内存是线程私有的。JVM 中的寄存器类似于物理寄存器的一种抽象模拟，正如前面说的，它是线程私有的，所以生命周期与线程的生命周期保持一致。

根据 Java 虚拟机定义来看，程序寄存器区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

2.1.2 虚拟机栈

JVM 的架构是基于栈的，即程序指令的每一个操作都要经过入栈和出栈这样的组合型操作才能完成。

虚拟机栈是一种可以被用来快速访问的存储区域，该区域位于通用 RAM[2]里面，通过使用它的所谓的“栈指针”可以访问处理器[3]。栈是一种快速有效的分配存储方法，访问速度仅次于寄存器，堆栈指针若向下移动，则分配新的内存，若向上移动，则释放那些内存。由于 Java 虚拟机需要预先去生成相应的内存空间，所以当我们尝试运行程序的时候，Java 虚拟机必须知道被存储在栈内的所有数据的确切大小和生命周期，以便按照上面陈述的分配存储方法通过上下移动堆栈指针来动态调整内存空间。我们可以认为这一约束限制了程序的灵活性，所以这就是

---

[1] 即 CPU 分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。

[2] 随机存取存储器（Random Access Memory，RAM）又称作“随机存储器”，是与 CPU 直接交换数据的内部存储器，也叫主存（内存）。它可以随时读写，而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储媒介。

[3] 感觉像是设计时留了一个后门。

为什么只有某些 Java 数据，特别是对象引用，它被存储在栈里面，而应用程序内部数量庞大的 Java 对象没有被存储在虚拟机栈里面。

总的来说，栈的优势是访问速度比堆要快，它仅次于寄存器，并且栈数据是可以被共享的。栈的缺点是存储在栈里面的数据大小与生存期必须是确定的，从这一点来看，栈明显缺乏灵活性。虚拟机栈内主要被用来存放一些基本类型的变量，例如 int、short、long、byte、float、double、boolea、char，以及对象引用。

前面说过，虚拟机栈有一个很重要的特殊性，就是存放在栈内的数据可以共享。假设同时定义：

```
int a = 1;
int b = 1;
```

对于上面的代码，虚拟机处理第一条语句，首先它会在栈内创建一个变量为 a 的引用，然后查找栈内是否有 1 这个值，如果没找到，就将 1 存放进来，然后将 a 指向 1。接下来处理第二条语句，在创建完 b 的引用变量后，因为在栈内已经有 1 这个值，便将 b 直接指向 1。这样，就出现了 a 与 b 同时均指向 1 的情况。这时，如果存在第三条语句，它针对 a 再次定义为 a=4，那么编译器会重新搜索栈内是否有 4 值，如果没有，则将 4 存放进来，并令 a 指向 4，如果已经有了，则直接将 a 指向这个地址，因此 a 值的改变不会影响到 b 的值[1]。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享的方式存在明显的不同，因为这种情况 a 的修改并不会影响到 b，它是由虚拟机完成的，这样的做法有利于节省空间。而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

与程序计数器一样，Java 虚拟机栈也是线程私有的内存空间，它和 Java 线程在同一时间创建，它保存方法的局部变量、部分结果，并参与方法的调用和返回。

Java 虚拟机规范允许 Java 栈的大小是动态的或者是固定不变的。在 Java 虚拟机规范中定义了两种异常与栈空间有关，即 StackoverFlowError 和 OutofMemoryError。如果线程在计算过程中，请求的栈深度大于最大可用的栈深度，则程序运行过程中会抛出 StackoverFlowError 异常。如果 Java 栈可以动态扩展，而在扩展栈的过程中没有足够的内存空间来支持栈的发展，则程序运行过程中会抛出 OutofMemoryError 异常。

我们可以使用-XSS 选项来设置虚拟机栈的大小，栈的大小直接决定了函数调用的最大可达深度。

---

[1] 可以理解为 a 和 b 两个人可以住在一个墙门里，也可以分住两个墙门。

代码清单 2-1 所示代码展示了一个递归调用的应用。计数器 COUNT 记录了递归的层次，recusion 方法是一个没有出口的递归函数，通过 testStack 方法的调用，该函数会不断地申请栈的深度，最终程序一定会导致虚拟机栈溢出。为了方便记录测试数据，程序会在第 13 行代码中当栈溢出异常发生时打印出虚拟机栈的当前深度。

代码清单 2-1　虚拟机栈示例代码 TestJVMStack

```
public class TestJVMStack {
    private int count = 0;
    //没有出口的递归函数
    public void recursion(){
    count++;//每次调用深度加 1
    recursion();//递归
    }

    public void testStack(){
    try{
        recursion();
    }catch(Throwable e){
        System.out.println("deep of stack is "+count);//打印栈溢出的深度
        e.printStackTrace();
    }
    }
    public static void main(String[] args){
    TestStack ts = new TestStack();
    ts.testStack();
    }
}
```

代码清单 2-1 程序输出如代码清单 2-2 所示，笔者本机运行的程序最终在申请到 9013[1]层虚拟机栈时，抛出异常。

代码清单 2-2　代码清单 2-1 程序运行输出

```
Java.lang.StackOverflowError
   at TestStack.recursion(TestStack.Java:7)
   at TestStack.recursion(TestStack.Java:7)
   at TestStack.recursion(TestStack.Java:7)
   at TestStack.recursion(TestStack.Java:7)
```

[1] 注意每台电脑都会输出不一样的结果，这是因为计算机本身配置，例如 CPU、内存，都会存在不同。

```
    at TestStack.recursion(TestStack.Java:7)
    at TestStack.recursion(TestStack.Java:7)
    at TestStack.recursion(TestStack.Java:7)deep of stack is 9013
```

如果系统需要支持更深的栈调用，则可以使用选项-Xss1M 运行程序，可以扩大虚拟机栈的大小，刚才的配置扩大栈空间的最大值为 1MB。

再来深入讨论虚拟机栈的内部结构。如图 2-2 所示，虚拟机栈在运行时使用一种叫作栈帧的数据结构保存上下文数据，栈帧里面存放了方法的局部变量表、操作数栈、动态连接方法和返回地址等信息。每一个方法的调用都伴随着栈帧的入栈操作，相应地，方法的返回则表示栈帧的出栈操作。如果方法调用时，方法的参数和局部变量相对较多，那么栈帧中的局部变量表就会比较大，栈帧会不断膨胀以满足方法调用所需传递的信息增大需求。因此，单个方法调用所需的栈空间也会比较多。

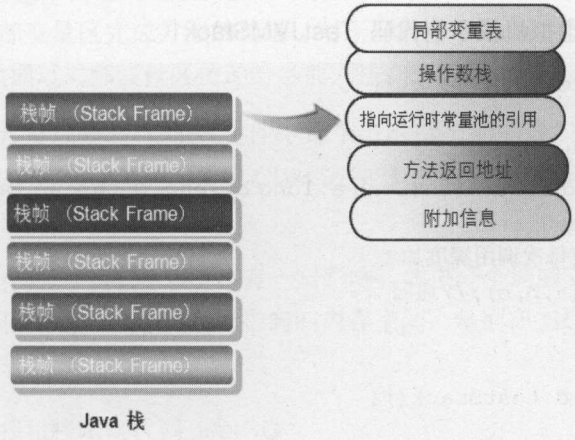


图 2-2　虚拟机栈引用图

栈帧由三部分组成，即局部变量区（Local Variables）、操作数栈（Operand Stack）和帧数据区（Frame Data）。

局部变量区被定义为一个从 0 开始的数字数组，byte、short、char 在存储前被转换为 int，boolean 也被转换为 int，0 表示 false，非 0 表示 true，long 和 double 则占据两个字长。注意，局部变量区是通过数组下标访问的。

操作数栈也被组织为一个数字数组，但不同于局部变量区，它不是通过数组下标访问的，

而是通过栈的 Push 和 Pop[1]操作，前一个操作 Push 进的数据可以被下一个操作 Pop 出来使用。

帧数据区这部分的作用主要有三点。

- 解析常量池里面的数据。
- 方法执行完后处理方法返回，恢复调用方现场。
- 方法执行过程中抛出异常时的异常处理，存储在一个异常表，当出现异常时虚拟机查找相应的异常表看是否有对应的 Catch 语句，如果没有就抛出异常终止这个方法调用。

函数嵌套调用的次数由栈的大小决定。一般来说，栈越大，函数嵌套调用次数越多。对一个函数而言，它的参数越多，内部局部变量越多，它的栈帧就越大，其嵌套调用次数就会减少。

代码清单 2-3 所示代码相较于代码清单 2-1 代码，增加了 recursion 方法的参数，用以展示内部局部变量增多之后的变化。

代码清单 2-3　虚拟机栈示例代码 TestJVMStack1

```
public class TestJVMStack1 {
    private int count = 0;
    //没有出口的递归函数
    public void recursion(long a,long b,long c) throws InterruptedException{
    long d=0,e=0,f=0;
    count++;//每次调用深度加 1
    recursion(a,b,c);//递归
}

    public void testStack(){
    try{
        recursion(1L,2L,3L);
    }catch(Throwable e){
        System.out.println("deep of stack is "+count);//打印栈溢出的深度
        e.printStackTrace();
    }
    }

    public static void main(String[] args){
    TestStack ts = new TestStack();
```

[1] 这是栈的两个操作组合，即进栈、出栈。

```
        ts.testStack();
        }
}
```

代码清单 2-3 代码运行后的输出如代码清单 2-4 所示。

代码清单 2-4　代码清单 2-3 运行输出

```
deep of stack is 3432
Java.lang.StackOverflowError
at TestStack.recursion(TestStack.Java:8)
```

从代码清单 2-4 我们可以看到，随着代码传入了多个参数和局部变量，栈帧大小就会膨胀。

在栈帧中，与性能调优关系最为密切的部分就是前面提到的局部变量区。局部变量区被用于存放方法的参数和进行方法内部的引用。局部变量区以“字”为单位进行内存的划分，一个字为 32 位长度。对于 long 和 double 型的变量，则占用 2 个字，其余类型使用 1 个字。在方法执行时，虚拟机使用局部变量区完成方法的传递，对于非静态（static）方法，虚拟机还会将当前对象（this）作为参数通过局部变量区传递给当前方法。

使用 JClassLib[1]工具可以查看 Class 文件中每个方法所分配的最大局部变量区的容量。JClassLib 工具是开源软件，它可以用于查看 Class 文件的结构，包括常量池、接口、属性、方法，还可以用于查看文件的字节码。

局部变量区的基本单位“字”对 GC[2]也有一定影响。如果一个局部变量被保存在局部变量区里面，那么 GC 能引用到这个局部变量所指向的内存空间，从而在 GC 时无法回收这部分空间。如代码清单 2-5 程序所示，演示 GC 参与。

代码清单 2-5　虚拟机栈示例代码 testGC

```
public class testGC {
    public static void test1()
    {
        byte[] a = new byte[6*1024*1024];
    }
    System.gc();
    System.out.println("first explict gc over");
```

[1] JClassLib 不但是一个字节码阅读器，而且还包含一个类库允许开发者读取、修改、写入 Java Class 文件与字节码。

[2] 即 Garbage Collection。

```
    public static void main(String[] args){
    testGC.test1();
    }
}
```

代码清单 2-5 中，第 4 行定义了一个局部变量 a，并且它的作用范围仅限于大括号中。在显式的 GC 调用时，变量 b 已经超过了它的作用范围，其对应的堆空间理应被回收。而事实上，由于变量 a 仍在该栈帧的局部变量区内，因此 GC 可以引用到该内存块，进而阻碍了其回收过程。

假设在该变量失效后，在这个函数体内又未能定义足够多的局部变量来复用该变量所占的基本单位“字”，那么在整个函数体内部，这块内存区域是不会被 GC 回收的。如果函数体内的后续操作非常费时或者又申请了较大的内存空间，则对系统性能将会造成较大的压力。在这种环境下，可以通过手动给要释放的变量赋值为 Null 的方法来解决这个潜在的性能问题。

2.1.3 本地方法栈

我们知道，当 C 语言编写的程序调用一个 C 函数时，其栈操作都是确定的，首先传递给该函数的参数以某个确定的顺序被压入栈，然后它的返回值也以确定的方式被传回给调用者。同样，这就是虚拟机实现本地方法栈的方式，很可能本地方法接口需要回调 Java 虚拟机中的 Java 方法（这也是由设计者决定的），在这种情形下，该线程会保存本地方法栈的状态并进入到另一个 Java 栈。就像其他运行时内存区一样，本地方法栈占用的内存区也不需要是固定大小的，它可以根据需要动态扩展或者收缩，某些 JVM 也允许用户或者程序员指定该内存区的初始大小以及最大、最小值。

本地方法栈（Native Method Stacks）和 Java 虚拟机栈的功能很相似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈用于管理本地方法的调用。本地方法并不是用 Java 实现的，而是使用 C 实现的。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界，本地方法可以通过本地方法接口来访问虚拟机内部的运行时数据区，但不止于此，它还可以做任何它想做的事情。比如，它甚至可以直接使用本地处理器中的寄存器，或者直接从本地内存的堆中分配任意数量的内存。总之，它和虚拟机拥有同样的权限（或者说能力）。

本地方法本质上是依赖于实现的，虚拟机实现的设计者可以自由地决定使用怎样的机制来让 Java 程序调用本地方法，任何本地方法接口都会使用某种本地方法栈。当线程调用 Java 方法时，虚拟机会创建一个新的栈帧并压入 Java 栈，然而当他调用的是本地方法时，虚拟机会保持 Java 栈不变，不再在线程的 Java 栈内压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。可以把这个做法看作虚拟机利用本地方法来动态扩展自己，就如同 Java 虚拟机的

实现是按照其中运行的 Java 程序的顺序，调用属于虚拟机内部的另一个（动态连接的）方法。

注意，在 SUN 的 HotSpot 虚拟机中，不区分本地方法栈和虚拟机栈。因此，和虚拟机栈一样，它也会抛出 STACKOVERFLOWERROR 和 OUTOFMEMORYERROR。

2.1.4　Java 堆

堆在 JVM 规范里是一种通用性的内存池（也存在于 RAM 中），用于存放所有的 Java 对象。堆是一个运行时数据区，类的对象从中分配空间，这些对象通过 New 关键字建立，它们不需要程序代码来显式地释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存周期也不需要事先告诉编译器。由于它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走那些不再使用的数据。但缺点是，由于要在运行时动态分配内存，所以数据访问速度较慢。大多数的虚拟机里，Java 中的对象和数组都存放在堆中。

堆不同于栈的优势是，虚拟机不需要知道要从堆内分配多少存储区域，也不必知道存储的数据在堆内需要存活多长时间。因此，在堆内分配存储相较于栈来说，有很大的灵活性。当你需要创建一个对象的时候，只需要引用 New 关键字写一行简单的代码，当执行这行代码时，会自动在堆内进行存储分配。当然，为这种灵活性必须要付出相应的代价，即用堆进行存储分配比用栈进行存储需要更多的时间。

Java 堆区在 JVM 启动的时候即被创建，它只要求逻辑上是连续的，在物理空间上可以是不连续。所有的线程共享 Java 堆，在这里可以划分线程私有的缓冲区（Thread Local Allocation Buffer，TLAB）[1]。

如前所述，Java 堆区是一块用于存储对象实例的内存区，同时也是 GC（Garbage Collection，垃圾收集器）执行垃圾回收的重点区域。正是因为 Java 堆区是 GC 的重点回收区域，所以 GC 极有可能会在大内存的使用和频繁进行垃圾回收过程上成为系统性能瓶颈。为了解决这个问题，JVM 的设计者们开始考虑是否一定需要将对象实例存储到 Java 堆区内。基于 OpenJDK[2]深度定制的 TaobaoJVM[3]，其中创新的 GCIH（GC invisible heap）技术实现了 off-heap，即将生命周期较长的 Java 对象从 heap 中移到 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，以此达

---

[1] 第 1 章有定义介绍。

[2] OpenJDK 作为 GPL 许可（GPL-licensed）的 Java 平台的开源化实现，Sun 正式发布它已经六年有余。从发布那一时刻起，Java 社区的大众就又开始努力学习，以适应这个新的开源代码基础（code-base）。

[3] 由 AliJVM 团队发布，是 AliJVM 团队基于 OpenJDK HotSpot VM 发布的国内第一个优化、定制且开源的服务器版 Java 虚拟机。目前已经在淘宝、天猫上线，把 Oracle 官方 JVM 版本全部替换了。

到降低 GC 的回收频率和提升 GC 的回收效率的目的。

除此之外，逃逸分析（图 2-3）与栈上分配这样的优化技术同样也是降低 GC 回收频率和提升 GC 回收效率的有效方式。这样一来，Java 堆区就不再是 Java 对象内存分配的唯一选择了。目前主流的垃圾收集算法是按代（Generation）收集，即按照对象的生存时间分为年轻代和老年代。年轻代又进一步被划分为 Eden 区、From Survivor 区和 To Survivor 区，主要是为了垃圾回收用途。

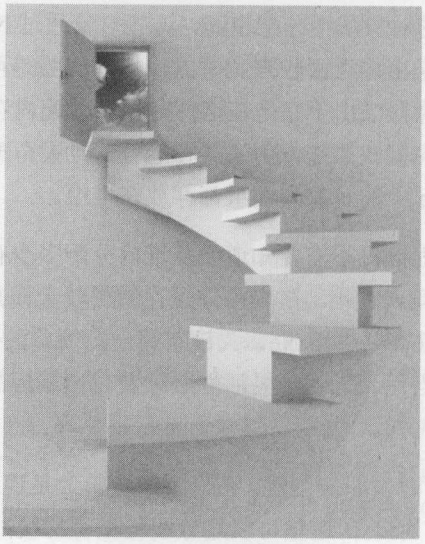


图 2-3　逃逸分析示意图

逃逸分析英文全名是 Escape Analysis。在计算机语言编译器优化原理中，逃逸分析是指分析指针动态范围的方法，它同编译器优化原理的指针分析和外形分析相关联。计算机软件方面，逃逸分析指的是计算机语言编译器语言优化管理中，分析指针动态范围的方法。通俗点讲，如果一个对象的指针被多个方法或线程引用时，那我们可以称这个指针发生了逃逸。Java 语言也有逃逸情况存在，示例代码如代码清单 2-6 所示。

代码清单 2-6　逃逸分析示例代码 escapeAnalysisClass

```
public class escapeAnalysisClass{
    public static B b;

    public void globalVariablePointerEscape(){//给全局变量赋值，发生逃逸
        b=new B();
    }
```

```
    public B methodPointerEscape(){//方法返回值，发生逃逸
        return new B();
    }

    public void instancePassPointerEscape(){
        methodPointerEscape().printClassName(this);//实例引用发生逃逸
    }


}

class B{
    public void printClassName(G g){
        System.out.println(g.getClass().getName());
    }
}
public class G {
    public static B b;
    public void globalVariablePointerEscape(){//给全局变量赋值，发生逃逸
        b=new B();
    }
    public B methodPointerEscape(){//方法返回值，发生逃逸
        return new B();
    }
    public void instancePassPointerEscape(){
        methodPointerEscape().printClassName(this);//实例引用发生逃逸
    }
 }
```

代码清单 2-6 所示的这个例子中，一共举了 3 种常见的指针逃逸场景，分别是全局变量赋值、方法返回值、实例引用传递。

逃逸的分析研究对于 Java 虚拟机有什么好处？Java 对象总是在堆中被分配的，因此 Java 对象的创建和回收对系统的开销是很大的。Java 语言被批评的一个地方，也是被认为 Java 性能慢的一个原因就是 Java 不支持运行时栈分配对象，缺少像 C#里面的值对象或者 C++里面的 struct 结构。JDK6 里的 Swing 内存和性能消耗的瓶颈就是由于发生逃逸所造成的。栈里面只保存了对象的指针，当对象不再被使用后，需要依靠 GC 来遍历引用树并回收内存，如果对象数量较多，会给 GC 带来较大压力，也直接或间接地影响了应用的性能。减少临时对象在堆内分配的数量，无疑是最有效的优化方法。

在 Java 应用里普遍存在一种场景，一般是在方法体内，声明了一个局部变量，且该变量在方法执行生命周期内未发生逃逸，因为在方法体内未将引用暴露给外面，按照 JVM 内存分配机制，首先会在堆内创建变量类的实例，然后将返回的对象指针压入调用栈，继续执行。这是 JVM 优化前的方式。

我们可以采用逃逸分析原理对 JVM 进行优化，即针对栈的重新分配方式。首先需要分析并且找到未逃逸的变量，将变量类的实例化内存直接在栈里分配（无须进入堆），分配完成后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。通过这种优化方式，与优化前的方式的主要区别在于栈空间直接作为临时对象的存储介质，从而减少了临时对象在堆内的分配数量。

基于逃逸分析的 JVM 优化原理很简单，但是在应用过程中还是有诸多因素需要被考虑。比如，由于与 Java 的动态性有冲突，所以逃逸分析不能在静态编译时进行，必须在 JIT[1] 里完成。因为你可以在运行时通过动态代理改变一个类的行为，此时，逃逸分析是无法得知类已经变化了。

那么 JIT 怎么通过逃逸分析进行代码优化呢？如代码清单 2-7 所示。

代码清单 2-7　逃逸分析优化示例代码 my_method

```
public void my_method(){
    V v=new V();
    //use v
    ......
    v=null;
}
```

在这个方法中创建的局部对象被赋给了 v，但是没有返回，没有赋给全局变量等操作，因此这个对象是没有逃逸的，是可以在运行时在栈上进行分配和销毁的对象。没有发生逃逸的对象，由于生命周期都在一个方法体内，因此它们可以在运行时在栈上分配并销毁。

这样在 JIT 编译 Java 伪代码时，如果能分析出这种代码，那么非逃逸对象其创建和回收就可以在栈上进行，从而能大大提高 Java 的运行性能。

另外，为什么要在逃逸分析之前进行内联分析呢？这是因为往往有些对象在被调用过程中

[1] JIT（just in time）编译器。当 Java 执行 runtime 环境时，每遇到一个新的类，JIT 编译器在此时就会针对这个类进行编译作业。经过编译后的程序，被优化成相当精简的二进制，这种程序的执行速度相当快。

创建并返回给调用过程，调用过程使用完该对象就被销毁了。这种情况下如果将这些方法进行内联，它们就由两个方法体变成一个方法体了，这种原来通过返回传递的对象就变成了方法内的局部对象，就变成了非逃逸对象了，这样这些对象就可以在同一栈上进行分配了。逃逸优化示意图如图 2-4 所示。

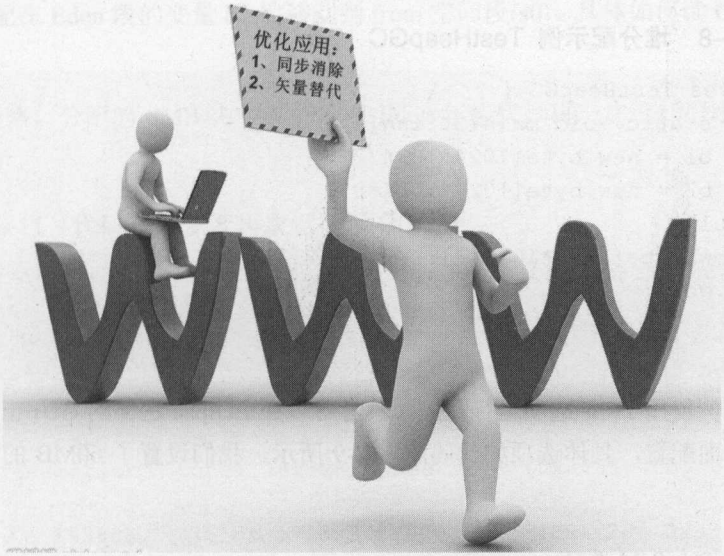


图 2-4　逃逸优化示意图

Java7 开始支持对象的栈分配和逃逸分析机制。这样的机制除了能将堆分配对象变成栈分配对象以外，逃逸分析还有其他两个优化应用。

- 同步消除。线程同步的代价是相当高的，同步的后果是降低并发性和性能。逃逸分析可以判断出某个对象是否始终只被一个线程访问，如果只被一个线程访问，那么对该对象的同步操作就可以转化成没有同步保护的操作，这样就能大大提高并发性和性能。
- 矢量替代。逃逸分析方法如果发现对象的内存存储结构不需要连续进行的话，就可以将对象的部分甚至全部都保存在 CPU 寄存器内，这样能大大加快访问速度。

Java7 完全支持栈式分配对象，JIT 支持逃逸分析优化，此外 Java7 还默认支持 OpenGL 的加速功能。

几乎所有的对象和数组都是在堆中分配空间的。Java 堆由年轻代和老年代两个部分组成，年轻代用于存放刚刚产生的对象和年轻的对象，如果对象一直没有被回收，生存得足够长，对象就被移入老年代。年轻代又可进一步细分为 Eden、Survivor Space0 和 Survivor Space1。Eden

即对象的出生地，大部分对象刚刚建立时都会被存放在这里。Survivor 空间是存放其中至少经历了一次垃圾回收，并得以幸存下来的对象。如果在幸存区的对象到了指定年龄仍未被回收，则有机会进入老年代。

对象在内存中的分配方式实例代码如代码清单 2-8 所示。

代码清单 2–8　堆分配示例 TestHeapGC

```
public class TestHeapGC {
    public static void main(String[] args){
    byte[] b1 = new byte[1024*1024/2];
    byte[] b2 = new byte[1024*1024*8];
    b2 = null;
    b2 = new byte[1024*1024*8];//进行一次年轻代 GC
    System.gc();
  }
}
```

我们针对本例使用 JVM 选项运行程序，读者可以在 Eclipse 这样的 GUI 工具里面设置，也可以在命令行里面配置，具体选项如代码清单 2-9 所示，我们设置了 60MB 的内存空间作为堆空间。

代码清单 2–9　JVM 选项

```
-XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15
-Xms40M -Xmx40M -Xmn20M
```

代码清单 2-8 的代码采用代码清单 2-9 的配置后运行，GC 输出日志如代码清单 2-10 所示。

代码清单 2–10　代码清单 2–8 运行后 GC 输出

```
[GC [DefNew: 9031K->661K(18432K), 0.0022784 secs] 9031K->661K(38912K),
0.0023178 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
Heap
 def new generation   total 18432K, used 9508K [0x34810000, 0x35c10000,
0x35c10000)
eden space 16384K,  54% used [0x34810000, 0x350b3e58, 0x35810000)
  from space 2048K,  32% used [0x35a10000, 0x35ab5490, 0x35c10000)
  to   space 2048K,   0% used [0x35810000, 0x35810000, 0x35a10000)
 tenured generation   total 20480K, used 0K [0x35c10000, 0x37010000, 0x37010000)
   the space 20480K,   0% used [0x35c10000, 0x35c10000, 0x35c10200,
0x37010000)
 compacting perm gen  total 12288K, used 374K [0x37010000, 0x37c10000,
```

```
0x3b010000)
      the space 12288K,   3% used [0x37010000, 0x3706db10, 0x3706dc00, 0x37c10000)
       ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
       rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

从 GC 输出日志可以看出，在进行多次内存分配的过程中，触发了一次年轻代 GC。在这次 GC 中，原本分配在 Eden 段的变量 b1 被移动到 from 空间段(s0)。具体如何读 GC 输出，会在第 3 章进一步解释。

我们调整策略，分配的 8MB 内存被分配在 Eden 年轻代。再一次运行程序，GC 输出如代码清单 2-11 所示。

代码清单 2-11　代码清单 2-8 再次运行后 GC 输出

```
   [GC [DefNew: 9031K->661K(18432K), 0.0023186 secs] 9031K->661K(38912K),
0.0023597 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
   [Full GC (System) [Tenured: 0K->8853K(20480K), 0.0179368 secs]
9180K->8853K(38912K), [Perm : 374K->374K(12288K)], 0.0179893 secs] [Times:
user=0.00 sys=0.02, real=0.02 secs]
   Heap
    def new generation   total 18432K, used 327K [0x34810000, 0x35c10000,
0x35c10000)
     eden space 16384K,   2% used [0x34810000, 0x34861f28, 0x35810000)
     from space 2048K,   0% used [0x35a10000, 0x35a10000, 0x35c10000)
     to   space 2048K,   0% used [0x35810000, 0x35810000, 0x35a10000)
    tenured generation   total 20480K, used 8853K [0x35c10000, 0x37010000,
0x37010000)
      the space 20480K,  43% used [0x35c10000, 0x364b5458, 0x364b5600, 0x37010000)
    compacting perm gen  total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
      the space 12288K,   3% used [0x37010000, 0x3706db40, 0x3706dc00, 0x37c10000)
       ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
       rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

代码清单 2-11 的输出显示，在 Full GC 之后，年轻代空间被清空，未被回收的对象全部被移入老年代。

2.1.5　方法区

方法区主要保存的信息是类的元数据。方法区与堆空间类似，它也是被 JVM 中所有的线程共享的区域。如图 2-5 所示，方法区中最为重要的是类的类型信息、常量池、域信息、方法信

息。类型信息包括类的完整名称、父类的完整名称、类型修饰符（public/protected/private）和类型的直接接口类表。

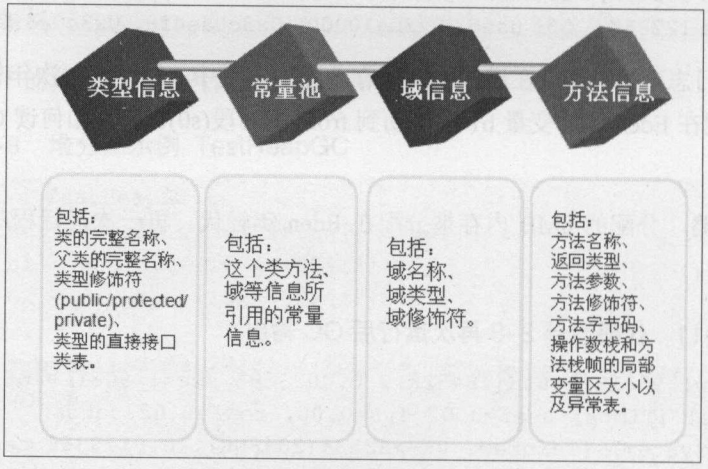


图 2-5　方法区组成部分图

常量池包括类方法、域等信息所引用的常量信息。域信息包括域名称、域类型和域修饰符。方法信息包括方法名称、返回类型、方法参数、方法修饰符、方法字节码、操作数栈和方法栈帧的局部变量区大小以及异常表。方法区是线程间共享的，当两个线程同时需要加载一个类型时，只有一个类会请求 ClassLoader[1]加载，另一个线程则会等待。总而言之，方法区内保存的信息大部分来自于 Class 文件，是 Java 应用程序运行必不可少的重要数据。

在 HotSpot[2]虚拟机中，方法区也被称为永久区，是一块独立于 Java 堆的内存空间。虽然被叫作永久区，但是在永久区中的对象同样也是可以被 GC 回收的，只是对于 GC 的对应策略与 Java 堆空间略有不同。

GC 针对永久区的回收，通常主要从两个方面分析：一是 GC 对永久区常量池的回收，二是永久区对类元数据的回收。

HotSpot 虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引

---

[1] 即类加载器，用来加载 Java 类到 Java 虚拟机中。与普通程序不同的是。Java 程序（Class 文件）并不是本地的可执行程序。当运行 Java 程序时，首先运行 JVM（Java 虚拟机），然后再把 Java Class 加载到 JVM 里头运行，负责加载 Java Class 的这部分就叫作 ClassLoader。

[2] Oracle 公司收购 SUN 公司后整合了原有多种 JVM 技术之后推出的一种 JVM 实现技术，相对以往的 JVM，在性能和扩展能力上都得到了很大的提升，因此它不是一个独立产品，可以理解为 Sun（Oracle）实现的 JVM 版本的品牌商标。

用，就可以被回收。

代码清单 2-12 所示代码生成大量 String 对象，并将其加入常量池中。String.intern()方法的含义是：如果常量池中已经存在当前 String，则返回池中的对象；如果常量池中不存在当前 String 对象，则先将 String 加入常量池，并返回池中的对象引用。因此，不停地将 String 对象加入常量池会导致永久区饱和，如果 GC 不能回收永久区的这些常量数据，那么就会抛出 OutofMemoryError 错误。

代码清单 2-12　回收永久区示例 permGenGC

```
public class permGenGC {
    public static void main(String[] args){
    for(int i=0;i<Integer.MAX_VALUE;i++){
        String t = String.valueOf(i).intern();//加入常量池
    }
    }
}
```

同样的，JVM 设置如代码清单 2-13 所示。

代码清单 2-13　JVM 设置

```
-XX:PermSize=2M -XX:MaxPermSize=4M -XX:+PrintGCDetails
```

运行 2-12 程序后，GC 输出日志如代码清单 2-14 所示。

代码清单 2-14　GC 输出日志

```
   [Full GC [Tenured: 0K->149K(10944K), 0.0177107 secs] 3990K->149K(15872K),
[Perm : 4096K->374K(4096K)], 0.0181540 secs] [Times: user=0.02 sys=0.02,
real=0.03 secs]
   [Full GC [Tenured: 149K->149K(10944K), 0.0165517 secs] 3994K->149K(15936K),
[Perm : 4096K->374K(4096K)], 0.0169260 secs] [Times: user=0.01 sys=0.00,
real=0.02 secs]
   [Full GC [Tenured: 149K->149K(10944K), 0.0166528 secs] 3876K->149K(15936K),
[Perm : 4096K->374K(4096K)], 0.0170333 secs] [Times: user=0.02 sys=0.00,
real=0.01 secs]
```

从上面代码清单 2-14 的 GC 输出日志可以看出，每当常量池饱和时，Full GC 总能顺利回收常量池数据，确保程序稳定持续进行。

2.2 垃圾收集算法

目前有两种比较常见的垃圾标记算法，分别是引用计数算法和根搜索算法。引用计数器在微软的 COM 组件技术中、Adodb 的 ActionScript3 中都有使用。

2.2.1 引用计数法

引用计数法（Reference Counting）在 GC 执行垃圾回收之前，首先需要区分出内存中哪些是存活对象，哪些是已经死亡的对象。只有被标记为已经死亡的对象，GC 才会在执行垃圾回收时，释放掉其所占用的内存空间，因此这个过程我们可以称为垃圾标记阶段。

引用计数器的实现很简单，对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，则对象 A 就不可能再被使用。也就是说，引用计数器的实现只需要为每个对象配置一个整形的计数器即可。引用计数器算法的一大优势就是不用等待内存不够用的时候，才进行垃圾的回收，完全可以在赋值操作的同时检查计数器是否为 0，如果是的话就可以立即回收。

但是引用计数器有一个严重的问题，即无法处理循环引用的情况。一个简单的循环引用问题的描述如下：有对象 A 和对象 B，对象 A 中含有对象 B 的引用，对象 B 中含有对象 A 的引用。此时，对象 A 和对象 B 的引用计数器都不为 0，但是在系统中却不存在任何第 3 个对象引用了 A 或 B。也就是说，A 和 B 是应该被回收的垃圾对象，但由于垃圾对象间相互引用，从而使垃圾回收器无法识别，引起内存泄漏。

如图 2-6 所示，构造了一个列表，将最后一个元素的 next 属性指向第一个元素，即引用第一个元素，从而构成循环引用。这个时候如果将列表的头 head 赋值为 null，此时列表的各个元素的计数器都不为 0，同时也失去了对列表的引用控制，从而导致列表元素不能被回收。

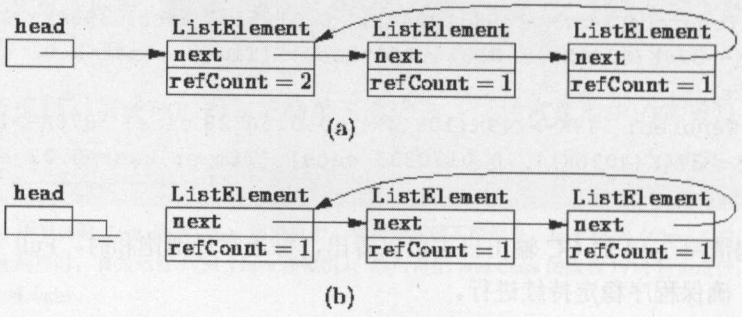


图 2-6 引用计数流程图

引用计数器拥有一些特性，首先它需要单独的字段存储计数器，这样的做法增加了存储空间的开销。其次，每次赋值都需要更新计数器，这增加了时间开销。再者，垃圾对象便于辨识，只要计数器为 0，就可作为垃圾回收。接下来它能方便及时地回收垃圾，没有延迟性。最后不能解决循环引用的问题。正是由于最后一条致命缺陷，导致在 Java 的垃圾回收器中没有使用这类算法。

2.2.2　根搜索算法

HotSpot 和大部分 JVM 都是使用根搜索算法作为垃圾标记的算法实现。前面介绍过的引用计数算法尽管实现简单，执行效率也不错，但是该算法本身却存在一个较大的弊端，甚至会影响到垃圾标记的准确性。由于引用计数算法会为程序中的每一个对象都创建一个私有的引用计数器，当目标对象被其他存活对象引用时，引用计数器中的值则会加 1，不再引用时便会减 1，当引用计数器中的值为 0 的时候，就意味着该对象已经不再被任何存活对象引用，可以被标记为垃圾对象。采用这种方式看起来似乎没有任何问题，但是如果一些明显已经死亡了的对象尽管没有被任何的存活对象引用，但是它们彼此之间却存在相互引用时，引用计数器中的值则永远不会为 0，这样便会导致 GC 在执行内存回收时永远无法释放掉这种无用对象所占用的内存空间，极有可能引发内存泄漏。

相对于引用计数算法而言，根搜索算法不仅同样具备实现简单和执行高效等特点，更重要的是该算法可以有效地解决在引用计数算法中一些已经死亡的对象因相互引用而导致的无法正确被标记的问题，防止内存泄漏的发生。简单来说，根搜索算法是以根对象集合为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达（使用根搜索算法后，内存中的存活对象都会被根对象集合直接或间接连接着），如果目标对象不可达，就意味着该对象已经死亡，便可以在 instanceOopDesc[1]的 Mark World 中将其标记为垃圾对象。在根搜索算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。在 HotSpot 中，根对象集合中包含了 5 个元素，Java 栈内的对象引用、本地方法栈内的对象引用、运行时常量池中的对象引用、方法区中类静态属性的对象引用以及与一个类对应的唯一数据类型的 Class 对象。

注意，在根搜索算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是

---

[1] HotSpot 在 C++代码中用 instanceOopDesc 类来表示 Java 对象，而该类继承 oopDesc，所以 HotSpot 中的 Java 对象也自然拥有 oopDesc 所声明的头部。

此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法，或者 finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。如果这个对象被判定为有必要执行 finalize()方法，那么这个对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象在 finalize()方法中执行缓慢，或者发生了死循环（更极端的情况），很可能会导致 F-Queue 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this 关键字）赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合。如果对象这时候还没有逃脱，那它就真的离死不远了。从代码清单 2-15 中可以看到一个对象的 finalize()被执行，但是它仍然可以存活。

代码清单 2-15　逃脱回收实验

```
public class FinalizeEscapeGC {
    public static FinalizeEscapeGC SAVE_HOOK = null;
    public void isAlive() {
        System.out.println("yes, i am still alive");
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize mehtod executed!");
        FinalizeEscapeGC.SAVE_HOOK = this;
    }

    public static void main(String[] args) throws Throwable {
        SAVE_HOOK = new FinalizeEscapeGC();
        //对象第一次成功拯救自己
        SAVE_HOOK = null;
        System.gc();
        // 因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
        Thread.sleep(500);
        if (SAVE_HOOK != null) {
            SAVE_HOOK.isAlive();
        } else {
```

```
            System.out.println("no, i am dead");
        }
        // 下面这段代码与上面的完全相同，但是这次自救却失败了
        SAVE_HOOK = null;
        System.gc();
        // 因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
        Thread.sleep(500);
        if (SAVE_HOOK != null) {
            SAVE_HOOK.isAlive();
        } else {
            System.out.println("no, i am dead");
        }
    }
}
```

输出如代码清单 2-16 所示。

代码清单 2–16　逃脱回收实验运行输出

```
finalize mehtod executed!
yes, i am still alive
no, i am dead
```

从代码清单 2-16 的运行结果可以看到，SAVE_HOOK 对象的 finalize()方法确实被 GC 收集器触发过，并且在被收集前成功逃脱了。另外一个值得注意的地方就是，代码中有两段完全一样的代码片段，执行结果却是一次逃脱成功，一次失败，这是因为任何一个对象的 finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize()方法不会被再次执行，因此第 2 段代码的自救行动失败了。

2.2.3　标记-清除算法（Mark-Sweep）

当成功区分出内存中存活对象和死亡对象后，GC 接下来的任务就是执行垃圾回收，释放掉无用对象所占用的内存空间，以便有足够的可用内存空间为新对象分配内存。目前在 JVM 中比较常见的三种垃圾收集算法是标记-清除算法（Mark-Sweep）、复制算法（Copying）、标记-压缩算法（Mark-Compact）。在介绍三种算法之前，我们先来通过表 2-1 看看它们之间的区别。

表 2-1 算法比较表

	Mark-Sweep	Mark-Compact	Copying
速度	中等	最慢	最快
空间开销	少（但会堆积碎片）	少（不堆积碎片）	通常需要活对象的2倍大小（不堆积碎片）
移动对象	否	是	是

标记-清除算法（Mark-Sweep）是一种非常基础和常见的垃圾收集算法，该算法被 J.McCarthy 等人在 1960 年提出并成功地发明并应用于 Lisp 语言。以餐巾纸作为示例，午餐过程中，餐厅里的所有人都根据自己的需要取用餐巾纸。当垃圾收集机器人想收集废旧餐巾纸的时候，它会让所有用餐的人先停下来，然后，依次询问餐厅里的每一个人：“你正在用餐巾纸吗？你用的是哪一张餐巾纸？”机器人根据每个人的回答将人们正在使用的餐巾纸画上记号。询问过程结束后，机器人在餐厅里寻找所有散落在餐桌上且没有记号的餐巾纸（这些显然都是用过的废旧餐巾纸），把它们统统扔到垃圾箱里。

回到算法本身。算法涉及几个概念，先来了解一下 mutator 和 collector，这两个名词经常在垃圾收集算法中出现，collector 指的就是垃圾收集器，而 mutator 是指除了垃圾收集器之外的部分，比如说我们的应用程序本身。mutator 的职责一般是 NEW（分配内存）、READ（从内存中读取内容）、WRITE（将内容写入内存），而 collector 则就是回收不再使用的内存来供 mutator 进行 NEW 操作的使用。mutator 根对象一般指的是分配在堆内存之外，可以直接被 mutator 直接访问到的对象，一般是指静态/全局变量以及 ThreadLocal 变量[1]。

标记-清除算法将垃圾回收分为两个阶段，标记阶段和清除阶段。在标记阶段，collector 从 mutator 根对象开始进行遍历，对从 mutator 根对象可以访问到的对象都打上一个标识，一般是在对象的 header 中，将其记录为可达对象。而在清除阶段，collector 对堆内存（heap memory）从头到尾进行线性的遍历，如果发现某个对象没有标记为可达对象，通过读取对象的 header 信息，则将其回收。一种可行的实现是，在标记阶段首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。

前面说过，标记-清除算法的执行过程分为“标记”和“清除”两大阶段。这种分步执行的思路奠定了现代垃圾收集算法的思想基础。与引用计数算法不同的是，标记-清除算法不需要运行环境监测每一次内存分配和指针操作，而只要在“标记”阶段中跟踪每一个指针变量的指向，用类似思路实现的垃圾收集器也常被后人统称为跟踪收集器（Tracing Collector）。

[1] 在 Java 中，存储在 Java.lang.ThreadLocal 中的变量和分配在栈上的变量方法内部的临时变量等都属于此类。

标记-清除算法最大的问题是存在大量的空间碎片，因为回收后的空间是不连续的。在对象的堆空间分配过程中，尤其是大对象的内存分配，不连续的内存空间的工作效率要低于连续的空间。

相对于另外两种内存回收算法而言，标记-清除算法（Mark-Sweep）不仅执行效率低下，更重要的是，由于被执行内存回收的无用对象所占用的内存空间有可能是一些不连续的内存块，不可避免地会产生一些内存碎片，从而导致后续没有足够的可用内存空间分配给较大的对象。

2.2.4　复制算法（Copying）

为了解决标记-清除算法在垃圾收集效率方面的缺陷，M.L.Minsky 于 1963 年发表了著名的论文，“一种使用双存储区的 Lisp 语言垃圾收集器（A LISP Garbage Collector Algorithm Using Serial Secondary Storage）”。M.L.Minsky 在该论文中描述的算法被人们称为复制算法，它也被 M.L.Minsky 本人成功地引入到了 Lisp 语言的一个实现版本中。

标记-清除算法后来被引入 JVM 中，提升了 GC 在垃圾标记和内存释放这两个阶段的执行效率。还是采用之前的餐厅示例，餐厅被垃圾收集机器人分成南区和北区两个大小完全相同的部分。午餐时，所有人都先在南区用餐（因为空间有限，用餐人数自然也将减少一半），用餐时可以随意使用餐巾纸。当垃圾收集机器人认为有必要回收废旧餐巾纸时，它会要求所有用餐者以最快的速度从南区转移到北区，同时随身携带自己正在使用的餐巾纸。等所有人都转移到北区之后，垃圾收集机器人只要简单地把南区中所有散落的餐巾纸扔进垃圾箱就算完成任务了。下一次垃圾收集的工作过程也大致类似，唯一的不同只是人们的转移方向变成了从北区到南区。如此循环往复，每次垃圾收集都只需简单地转移（也就是复制）一次，垃圾收集速度无与伦比——当然，对于用餐者往返奔波于南北两区之间的辛劳，垃圾收集机器人是绝不会流露出丝毫怜悯的。

回到算法本身。复制算法首先将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收。

如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量并不会太大。因此在真正需要垃圾回收的时刻，复制算法的效率是很高的。又由于对象在垃圾回收过程中统一被复制到新的内存空间中，因此，可确保回收后的内存空间是没有碎片的。该算法的缺点是将系统内存折半。

Java 的年轻代串行垃圾回收器中使用了复制算法的思想。年轻代分为 Eden 空间、From 空

间、To 空间 3 个部分。其中 From 空间和 To 空间可以视为用于复制的两块大小相同、地位相等，且可进行角色互换的空间块。From 和 To 空间也称为 Survivor 空间，即幸存者空间，用于存放未被回收的对象。

在垃圾回收时，Eden 空间中的存活对象会被复制到未使用的 Survivor 空间中（假设是 To），正在使用的 Survivor 空间（假设是 From）中的年轻对象也会被复制到 To 空间中（大对象，或者老年对象会直接进入老年代，如果 To 空间已满，则对象也会直接进入老年代）。此时，Eden 空间和 From 空间中的剩余对象就是垃圾对象，可以直接清空，To 空间则存放此次回收后的存活对象。这种改进的复制算法既保证了空间的连续性，又避免了大量的内存空间浪费。

基于分代的概念，Java 堆区如果还要更进一步细分的话，还可以划分为年轻代（YoungGen）和老年代（OldGen），其中年轻代又可以被划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。在 HotSpot 中，Eden 空间和另外两个 Survivor 空间默认所占的比例是 8∶1，当然开发人员可以通过选项“-XX:SurvivorRatio”调整这个空间比例。当执行一次 Minor GC（年轻代的垃圾回收）时，Eden 空间中的存活对象会被复制到 To 空间内，并且之前已经经历过一次 Minor GC 并在 From 空间中存活下来的对象如果还年轻的话同样也会被复制到 To 空间内。需要注意的是，在满足两种特殊情况下，Eden 和 From 空间中的存活对象将不会被复制到 To 空间内。首先是如果存活对象的分代年龄超过选项“-XX：MaxTenuringThreshold”所指定的阈值时，将会直接晋升到老年代中。其次当 To 空间的容量达到阈值时，存活对象同样也是直接晋升到老年代中。当所有的存活对象都被复制到 To 空间或者晋升到老年代后，剩下的均为垃圾对象，这就意味着 GC 可以对这些已经死亡了的对象执行一次 Minor GC，释放掉其所占用的内存空间。

当执行完 Minor GC 之后，Eden 空间和 From 空间将会被清空，而存活下来的对象则会被全部存储在 To 空间内，接下来 From 空间和 To 空间将会互换位置。其实复制算法无非就是使用 To Survivor 空间作为一个临时的空间交换角色，务必需要保证两块空间中一块必须是空的，这就是复制算法。尽管复制算法能够高效执行 Minor GC，但是它却并不适用于老年代中的内存回收，因为老年代中对象的生命周期都比较长，甚至在某些极端的情况下还能够与 JVM 的生命周期保持一致，所以如果老年代也采用复制算法执行内存回收，不仅需要额外的时间和空间，而且还会导致较多的复制操作影响到 GC 的执行效率。

总的来说，由于 JVM 中的绝大多数对象都是瞬时状态的，生命周期非常短暂，所以复制算法被广泛应用于年轻代中。分区、复制的思路不仅大幅提高了垃圾收集的效率，而且也将原本繁纷复杂的内存分配算法变得前所未有的简明和扼要（既然每次内存回收都是对整个半区的回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存就可以了），这简直是个奇迹！不过，任何奇迹的出现都有一定的代价，在垃圾收集技术中，复制

算法提高效率的代价是人为地将可用内存缩小了一半。

2.2.5　标记-压缩算法（Mark-Compact）

标记-清除算法的确可以应用在老年代中，但是该算法不仅执行效率低下，而且在执行完内存回收后还会产生内存碎片，所以 JVM 的设计者在此基础之上进行了改进，标记-压缩算法由此诞生。标记-压缩算法是标记-清除算法和复制算法的有机结合。把标记-清除算法在内存占用上的优点和复制算法在执行效率上的特长综合起来，这是所有人都希望看到的结果。不过，两种垃圾收集算法的整合并不像 1 加 1 等于 2 那样简单，必须引入一些全新的思路。

1970 年前后，G. L. Steele、C. J. Chene 和 D. S. Wise 等研究者陆续找到了正确的方向，标记-压缩算法的轮廓也逐渐清晰了起来。还是采用之前的餐厅示例，在我们熟悉的餐厅里，这一次，垃圾收集机器人不再把餐厅分成两个南北区域了。需要执行垃圾收集任务时，机器人先执行标记-清除算法的第一个步骤，为所有使用中的餐巾纸画好标记，然后，机器人命令所有就餐者带上有标记的餐巾纸向餐厅的南面集中，同时把没有标记的废旧餐巾纸扔向餐厅北面。这样一来，机器人只需要站在餐厅北面，怀抱垃圾箱，迎接扑面而来的废旧餐巾纸就行了。

回到算法本身。当成功标记出内存中的垃圾对象后，标记-压缩算法会将所有的存活对象都移动到一个规整且连续的内存空间中，然后执行 Full GC（老年代的垃圾回收，或者被称为 Major GC）回收无用对象所占用的内存空间。当成功执行压缩之后，已用和未用的内存都各自一边，彼此之间维系着一个记录下一次分配起始点的标记指针，当为新对象分配内存时，则可以使用指针碰撞（Bump the Pointer）技术修改指针的偏移量将新对象分配在第一个空闲内存位置上，为新对象分配内存带来便捷。

在 HotSpot 中，基于分代的概念，GC 所使用的内存回收算法必须结合年轻代和老年代各自的特点。简单来说，就是针对不同的代空间，从而结合使用不同的垃圾收集算法。为年轻代选择的垃圾收集算法通常是以速度优先，因为年轻代中所存储的瞬时对象生命周期非常短暂，可以有针对性地使用复制算法，因此执行 Minor GC 时，一定要保持高效和快速。而年轻代中的生存空间通常都比较小，所以回收年轻代时一定会非常频繁。但老年代通常使用更节省内存的回收算法，因为老年代中所存储的对象生命周期都非常长，并且老年代占据了大部分的堆空间，所以老年代的 Full GC 并不会跟年轻代的 Minor GC 一样频繁，不过一旦程序中发生一次 Full GC，将会耗费更长的时间来完成，那么在老年代中使用标记-清除算法或者标记-压缩算法执行垃圾回收将会是不错的选择。

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在年轻代经常

发生，但是在老年代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活的对象较多，复制的成本也将很高。标记-压缩算法是一种老年代的回收算法，它在标记-清除算法的基础上做了一些优化。也首先需要从根节点开始对所有可达对象做一次标记，但之后，它并不是简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

标记-压缩算法的总体执行效率高于标记-清除算法，又不像复制算法那样需要牺牲一半的存储空间，这显然是一种非常理想的结果。在许多现代的垃圾收集器中，人们都使用了标记-压缩算法或其改进版本。

2.2.6 增量算法（Incremental Collecting）

在垃圾回收过程中，应用软件将处于一种 Stop the World 的状态。在 Stop the World 状态下，应用程序所有的线程都会挂起，暂停一切正常的工作，等待垃圾回收的完成。如果垃圾回收时间过长，应用程序会被挂起很久，将严重影响用户体验或者系统的稳定性。为了解决这个问题，即对实时垃圾收集算法的研究直接导致了增量收集算法的诞生。

最初，为了进行实时垃圾收集，可以设计一个多进程的运行环境，比如用一个进程执行垃圾收集工作，另一个进程执行程序代码。这样一来，垃圾收集工作看上去就仿佛是在后台悄悄完成的，不会打断程序代码的运行。在收集餐巾纸的例子中，这一思路可以被理解为垃圾收集机器人在人们用餐的同时寻找废弃的餐巾纸并将它们扔到垃圾箱里。这个看似简单的思路会在设计和实现时碰上进程间冲突的难题。比如说，如果垃圾收集进程包括标记和清除两个工作阶段，那么，垃圾收集器在第一阶段中辛辛苦苦标记出的结果很可能被另一个进程中的内存操作代码修改得面目全非，以至于第二阶段的工作没有办法开展。

M. L. Minsky 和 D. E. Knuth 对实时垃圾收集过程中的技术难点进行了早期的研究，G. L. Steele 于 1975 年发表了题为“多进程整理的垃圾收集（*Multiprocessing Compactifying Garbage Collection*）”的论文，描述了一种被后人称为“Minsky-Knuth-Steele 算法”的实时垃圾收集算法。E.W.Dijkstra、L.Lamport、R.R.Fenichel 和 J.C.Yochelson 等人也相继在此领域做出了各自的贡献。1978 年，H.G.Baker 发表了“串行计算机上的实时表处理技术（*List Processing in Real Time on a Serial Computer*）”一文，系统阐述了多进程环境下用于垃圾收集的增量收集算法。

增量算法的基本思想是，如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域

的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

总的来说，增量收集算法的基础仍是传统的标记-清除和复制算法。增量收集算法通过对进程间冲突的妥善处理，允许垃圾收集进程以分阶段的方式完成标记、清理或复制工作。

2.2.7　分代收集算法（Generational Collecting）

1980 年前后，善于在研究中使用统计分析知识的技术人员发现，大多数内存块的生存周期都比较短，垃圾收集器应当把更多的精力放在检查和清理新分配的内存块上。还是餐巾纸的例子，如果垃圾收集机器人足够聪明，事先摸清了餐厅里每个人在用餐时使用餐巾纸的习惯——比如有些人喜欢在用餐前后各用掉一张餐巾纸，有的人喜欢自始至终攥着一张餐巾纸不放，有的人则每打一个喷嚏就去用一张餐巾纸——机器人就可以制定出更完善的餐巾纸回收计划，并总是在人们刚扔掉餐巾纸没多久就把垃圾捡走。这种基于统计学原理的做法当然可以让餐厅的整洁度成倍提高。D. E. Knuth、T. Knight、G. Sussman 和 R. Stallman 等人对内存垃圾的分类处理做了最早的研究。1983 年，H. Lieberman 和 C. Hewitt 发表了题为“基于对象寿命的一种实时垃圾收集器（*A Real-Time Garbage Collector Based on the Lifetimes of Objects*）”的论文。这篇著名的论文标志着分代收集算法的正式诞生。此后，在 H. G. Baker、R. L. Hudson、J. E. B. Moss 等人的共同努力下，分代收集算法逐渐成为了垃圾收集领域里的主流技术。

根据垃圾回收对象的特性，使用合适的算法回收，分代就是基于这种思想。它将内存区间根据对象的特点分成几块，根据每块内存区间的特点，使用不同的回收算法以提高垃圾回收的效率。

以 HotSpot 虚拟机为例，它将所有的新建对象都放入称为年轻代的内存区域，年轻代的特点是对象会很快回收，因此，在年轻代就选择效率较高的复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老年代的内存空间。在老年代中，几乎所有的对象都是经过几次垃圾回收后依然得以幸存的。因此，可以认为这些对象在一段时期内，甚至在应用程序的整个生命周期中，将是常驻内存的。如果依然使用复制算法回收老年代，将需要复制大量对象。再加上老年代的回收性价比也要低于年轻代，因此这种做法也是不可取的。根据分代的思想，可以对老年代的回收使用与年轻代不同的标记-压缩算法，以提高垃圾回收效率。

总的来说，分代收集算法是基于对对象生命周期分析后得出的垃圾回收算法。它把对象分为年轻代、老年代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进

行回收。JVM 垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

2.3 Garbage Collection

2.3.1 GC 概念

GC（Garbage Collection，垃圾收集器）就是 JVM 中自动内存管理机制的具体实现。在 HotSpot 中，GC 的工作任务主要可以划分为两大块，分别是内存的动态分配和垃圾回收。而在内存执行分配之前，GC 首先会对内存空间进行划分，考虑到 JVM 中存活对象的生命周期会具有两极化，因此应该采取不同的垃圾收集策略，分代收集由此诞生。目前几乎所有的 GC 都是采用分代收集算法执行垃圾回收的，所以 Java 堆区如果还要更进一步细分的话，还可以划分为年轻代（YoungGen）和老年代（OldGen），其中年轻代内又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。换句话说，内存空间究竟应该如何划分完全依赖于 GC 的设计。当内存空间划分完成后，GC 就可以为新对象分配内存空间，并区分出存储在内存中的对象哪些是存活的，哪些是已经死亡了的。如果对象已经死亡，那么就可以将其标记为垃圾。为了避免内存溢出，GC 就会释放掉无用对象所占用的内存空间，便于有足够的可用内存空间分配给新的对象实例。

一般来说当内存空间中的内存消耗达到了一定阈值的时候，GC 就会执行垃圾回收，而且回收算法必须非常精确，一定不能造成内存中存活的对象被错误地回收掉，也不能造成已经死亡的对象没有被及时地回收掉。而且 GC 执行内存回收的时候应该做到高效，不应该导致应用程序出现长时间的暂停，以及避免产生内存碎片。不过当 GC 执行垃圾回收时，不可避免地会产生一些内存碎片，因为被回收的内存空间极有可能是一些不连续的内存块，这样一来将会导致没有足够的连续可用内存分配给较大的对象，不过可以使用压缩算法消除内存碎片。

淘宝的技术团队对 Java 虚拟机的优化工作其实早已不是停留在简单的参数调整上面，而是充分结合了企业自身的业务特点以及实际的应用场景，在 OpenJDK 的基础之上通过修改大量的 HotSpot 源代码，深度定制了淘宝专属的高性能 Linux 虚拟机 TAOBAOVM。从严格意义上来说，在提升 Java 虚拟机性能的同时，严重依赖于物理 CPU 类型。也就是说，部署有 TAOBAOVM 的服务器中，CPU 全都是清一色的 Intel CPU，且编译手段采用的是 Intel C/CPP Compiler 进行编译，以此对 GC 性能进行提升。除了优化编译效果外，TAOBAOVM 还使用 crc32 指令实现 JVM intrinsic 降低 JNI 的调用开销。

除了在性能优化方面下足了功夫，TAOBAOVM 还在 HotSpot 的基础之上大幅度扩充了一

些特定的增强实现，比如创新的 GCIH（GC invisible heap）技术实现 off-heap，这样一来就可以将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，这样做最大的好处就是降低了 GC 的回收频率以及提升了 GC 的回收效率，并且 GCIH 中的对象还能够在多个 Java 虚拟机进程中实现共享。其他补充技术还有利用 PMU hardware 的 Java profiling tool 和诊断协助功能等。

在许多情况下，GC 不应该成为影响系统性能的瓶颈，可以根据以下六点来评估一款 GC 的性能。

- 吞吐量：程序的运行时间（程序的运行时间+内存回收的时间）。
- 垃圾收集开销：吞吐量的补数，垃圾收集器所占时间与总时间的比例。
- 暂停时间：执行垃圾收集时，程序的工作线程被暂停的时间。
- 收集频率：相对于应用程序的执行，收集操作发生的频率。
- 堆空间：Java 堆区所占的内存大小。
- 快速：一个对象从诞生到被回收所经历的时间。

2.3.2　垃圾收集器分类

由于 JDK 的版本处于高速迭代过程中，因此 Java 发展至今已经衍生了众多的 GC 版本，比如 Serial/Serial Old 收集器、ParNew 收集器、Parallel/Parallel Old 收集器、CMS（Concurrent-Mark-Sweep）收集器，以及从 JDK7 Update4 版本开始提供的 G1（Garbage-First）收集器等。

基于分代的概念，不同的分代空间中均活动着不同的 GC，比如 Serial 收集器就是一个典型的年轻代垃圾收集器，它采用复制算法回收年轻代无用对象的内存空间。当然，JVM 在实际运行过程中，年轻代和老年代中各自的 GC 需要组合在一起共同执行垃圾回收任务。如果年轻代的 GC 和老年代的 GC 相连，则意味着可以组合在一起使用，不过在实际开发过程中，年轻代和老年代的 GC 的组合方式还需要结合具体的应用场景进行分析后得到。

从不同角度分析垃圾收集器，可以将 GC 分为不同的类型。

按线程数分，可以分为串行垃圾回收器和并行垃圾回收器。

串行回收指的是在同一时间段内只允许一件事情发生，简单来说，当多个 CPU 可用时，也只能有一个 CPU 用于执行垃圾回收操作，并且在执行垃圾回收时，程序中的工作线程将会被暂停，当垃圾收集工作完成后才会恢复之前被暂停的工作线程，这就是串行回收。不过由于运行在客户端下的 Client 模式的 JVM 并没有低于暂停时间的要求，并且 Client 模式下的内存开销相

对于 Server 模式来说也更小，因此串行回收默认被应用在 Client 模式下的 JVM 中。和串行回收相反，并行收集可以运用多个 CPU 同时执行垃圾回收，因此提升了应用的吞吐量，不过并行回收仍然使用了“Stop-the-world”机制和复制算法。

串行收集器主要有两个特点。首先，它仅仅使用单线程进行垃圾回收。其次，它是独占式的垃圾回收方式。

在串行收集器进行垃圾回收时，Java 应用程序中的线程都需要暂停，等待垃圾回收的完成，这样给用户体验造成较差效果。虽然如此，串行收集器却是一个成熟、经过长时间生产环境考验的极为高效的收集器。年轻代串行处理器使用复制算法，实现相对简单，逻辑处理特别高效，且没有线程切换的开销。在诸如单 CPU 处理器或者较小的应用内存等硬件平台不是特别优越的场合，它的性能表现可以超过并行回收器和并发回收器。

并行收集器是工作在年轻代的垃圾收集器，它只简单地将串行回收器多线程化。它的回收策略、算法以及参数和串行回收器一样。

并行回收器也是独占式的回收器，在收集过程中，应用程序会全部暂停。但由于并行回收器使用多线程进行垃圾回收，因此，在并发能力比较强的 CPU 上，它产生的停顿时间要短于串行回收器，而在单 CPU 或者并发能力较弱的系统中，并行回收器的效果不会比串行回收器好，由于多线程的压力，它的实际表现很可能比串行回收器差。

按照工作模式分，可以分为并发式垃圾回收器和独占式垃圾回收器。并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间。独占式垃圾回收器（Stop the world）一旦运行，就停止应用程序中的其他所有线程，直到垃圾回收过程完全结束。

按碎片处理方式可分为压缩式垃圾回收器和非压缩式垃圾回收器。压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片。非压缩式的垃圾回收器不进行这步操作。

按工作的内存区间，又可分为年轻代垃圾回收器和老年代垃圾回收器。下面开始对它们进行逐一讨论。

2.3.3 Serial 收集器

Serial 收集器作用于年轻代中，它采用复制算法、串行回收和“Stop-the-World”机制的方式执行内存回收。在早期的 JDK 版本中，由于那个年代的 CPU 速度并没有这么快，所以在 CPU 受限于单个 CPU 的情况下，使用 Serial 收集器执行年轻代垃圾收集几乎是唯一的选择，并且

Serial 收集器默认也作为 HotSpot 中 Client 模式下的年轻代垃圾收集器。

除了年轻代之外，Serial 收集器还提供用于执行老年代垃圾收集的 Serial Old 收集器。Serial Old 收集器同样也采用了串行回收和“Stop-the-World”机制，只不过内存回收算法使用的是标记-压缩算法。在此需要注意的是，如果在 JVM 受限于单个 CPU 的环境下，使用 Serial 收集器加上 Serial Old 收集器的组合执行 Client 模式下的内存回收将会是不错的选择。基于串行回收的垃圾收集器适用于大多数对暂停时间要求不高的 Client 模式下的 JVM，由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销，虽然执行串行回收会降低程序的吞吐量，但是回收质量还是不错的。在程序中，开发人员可以通过选项“-XX:+UseSerialGC”手动指定使用 Serial 收集器执行内存回收任务。

可以把上面的内容总结如下。

1. 该算法的第一步是在老年代标记存活的对象。

2. 从头开始检查堆内存空间，并且只留下依然幸存的对象（清除）。

3. 最后一步，从头开始，顺序地填满堆内存空间，将存活的对象连续存放在一起，这样堆分成两部分，一边有存放的对象，一边没有对象（整理）。

4. Serial 收集器应用于小的存储器和少量的 CPU。

- **年轻代串行收集器**

在 HotSpot 虚拟机中，使用-XX：+UseSerialGC 参数可以指定使用年轻代串行收集器和老年代串行收集器。当 JVM 在 Client 模式下运行时，它是默认的垃圾收集器。

代码清单 2-17 所示是一次年轻代串行收集器的工作输出日志（使用-XX:+PrintGCDetails 开关）。

代码清单 2-17　年轻代串行收集器工作输出

```
[GC [DefNew: 3468K->150K(9216K), 0.0028638 secs][Tenured: 1562K->1712K
(10240K), 0.0084220 secs] 3468K->1712K(19456K), [Perm : 377K->377K(12288K)],
0.0113816 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

上面的输出显示了一次垃圾回收前的年轻代的内存占用量和垃圾回收后的年轻代内存占用量，以及垃圾回收所消耗的时间。

- **老年代串行收集器**

前面提起过，老年代串行收集器使用的是标记-压缩算法。和年轻代串行收集器一样，它也

是一个串行的、独占式的垃圾回收器。由于老年代垃圾回收通常会使用比年轻代垃圾回收更长的时间，因此，在堆空间较大的应用程序中，一旦老年代串行收集器启动，应用程序很可能会因此停顿几秒甚至更长时间。

虽然如此，老年代串行回收器可以和多种年轻代回收器配合使用，同时它也可以作为 CMS 回收器的备用回收器。

若要启用老年代串行回收器，可以尝试使用以下选项。

-XX:+UseSerialGC：年轻代、老年代都使用串行回收器。

输出如代码清单 2-18 所示。

代码清单 2-18　使用-XX:+UseSerialGC 选项

```
Heap
 def new generation   total 4928K, used 1373K [0x27010000, 0x27560000,
0x2c560000)
  eden space 4416K,  31% used [0x27010000, 0x27167628, 0x27460000)
  from space 512K,   0% used [0x27460000, 0x27460000, 0x274e0000)
  to   space 512K,   0% used [0x274e0000, 0x274e0000, 0x27560000)
 tenured generation   total 10944K, used 0K [0x2c560000, 0x2d010000,
0x37010000)
   the space 10944K,   0% used [0x2c560000, 0x2c560000, 0x2c560200,
0x2d010000)
 compacting perm gen  total 12288K, used 376K [0x37010000, 0x37c10000,
0x3b010000)
   the space 12288K,   3% used [0x37010000, 0x3706e0b8, 0x3706e200,
0x37c10000)
    ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000,
0x3ba10000)
    rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600,
0x3c610000)
```

也可以使用-XX:+UseParNewGC 选项，当使用它作为 JVM 选项时，指定年轻代使用并行收集器，老年代使用串行收集器。

2.3.4　ParNew 收集器

如果说 Serial 是年轻代中的单线程垃圾收集器，那么 ParNew 收集器则是 Serial 收集器的多线程版本。ParNew 收集器除了采用并行回收的方式执行内存回收外，两款垃圾收集器之间

几乎没有任何区别，因为ParNew收集器在年轻代中同样也是采用复制算法和“Stop-the-World”机制。

如果说 ParNew 收集器运行在多 CPU 的环境下，由于可以充分利用多 CPU、多核心等物理硬件资源优势，确实可以更快速地完成垃圾收集，提升程序的吞吐量，但是如果是在单个 CPU 的环境下，ParNew 收集器不见得比 Serial 收集器更高效。虽然 Serial 收集器是基于串行回收，但是由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销。所以从理论上来说，Serial 收集器的优势是在 JVM 受限于单 CPU 环境中，而 ParNew 收集器的优势则是体现在多 CPU、多核心的环境中，并且在某些注重低延迟的应用场景下，ParNew 收集器和 CMS（Concurrent-Mark-Sweep）收集器的组合模式，在 Server 模式下的内存回收效果很好。在程序中，开发人员可以通过选项“-XX:+UseParNewGC”手动指定使用 ParNew 收集器执行内存回收任务，如代码清单 2-19 所示。

代码清单 2-19　使用-XX:+UseParNewGC 选项

```
Heap
 par new generation   total 4928K, used 1373K [0x0f010000, 0x0f560000,
0x14560000)
  eden space 4416K,  31% used [0x0f010000, 0x0f167620, 0x0f460000)
  from space 512K,   0% used [0x0f460000, 0x0f460000, 0x0f4e0000)
  to   space 512K,   0% used [0x0f4e0000, 0x0f4e0000, 0x0f560000)
 tenured generation   total 10944K, used 0K [0x14560000, 0x15010000,
0x1f010000)
  the space 10944K,   0% used [0x14560000, 0x14560000, 0x14560200,
0x15010000)
 compacting perm gen  total 12288K, used 2056K [0x1f010000, 0x1fc10000,
0x23010000)
  the space 12288K,  16% used [0x1f010000, 0x1f2121d0, 0x1f212200,
0x1fc10000)
No shared spaces configured.
```

如果使用选项-XX:+UseParallelGC 配置 JVM，表示年轻代使用并行回收收集器，老年代使用串行收集器。输出如代码清单 2-20 所示。

代码清单 2-20　使用-XX:+UseParallelGC 选项

```
Heap
 PSYoungGen      total 4800K, used 1380K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 33% used [0x1dac0000,0x1dc19130,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
```

```
    to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
   PSOldGen        total 10944K, used 0K [0x13010000, 0x13ac0000, 0x1dac0000)
    object space 10944K, 0% used [0x13010000,0x13010000,0x13ac0000)
   PSPermGen       total 12288K, used 2056K [0x0f010000, 0x0fc10000,
0x13010000)
    object space 12288K, 16% used [0x0f010000,0x0f2121d0,0x0fc10000)
```

一次老年代串行回收器的工作输出如代码清单 2-21 所示。

代码清单 2-21　老年代串行回收器输出

```
  [Full GC [Tenured: 1712K->1699K(10240K), 0.0071963 secs] 1712K->1699K(19456K),
[Perm : 377K->372K(12288K)], 0.0072393 secs] [Times: user=0.00 sys=0.00, real=
0.01 secs]
```

上面的输出显示了垃圾回收前老年代和永久区的内存占用量，以及垃圾回收后老年代和永久区的内存使用量。

开启并行回收器可以使用选项-XX:+UseParNewGC，它表示年轻代使用并行收集器，老年代使用串行收集器。输出如代码清单 2-22 所示。

代码清单 2-22　使用–XX:+UseParNewGC 选项

```
  [GC [ParNew: 825K->161K(4928K), 0.0155258 secs][Tenured:
8704K->661K(10944K), 0.0071964 secs] 9017K->661K(15872K), [Perm :
2049K->2049K(12288K)], 0.0228090 secs] [Times: user=0.01 sys=0.00, real=0.01
secs]
  Heap
   par new generation   total 4992K, used 179K [0x0f010000, 0x0f570000,
0x14560000)
    eden space 4480K,   4% used [0x0f010000, 0x0f03cda8, 0x0f470000)
    from space 512K,   0% used [0x0f470000, 0x0f470000, 0x0f4f0000)
    to   space 512K,   0% used [0x0f4f0000, 0x0f4f0000, 0x0f570000)
   tenured generation   total 10944K, used 8853K [0x14560000, 0x15010000,
0x1f010000)
     the space 10944K,  80% used [0x14560000, 0x14e057c0, 0x14e05800,
0x15010000)
   compacting perm gen  total 12288K, used 2060K [0x1f010000, 0x1fc10000,
0x23010000)
     the space 12288K,  16% used [0x1f010000, 0x1f213228, 0x1f213400,
0x1fc10000)
  No shared spaces configured.
```

2.3.5　Parallel 收集器

就目前而言，HotSpot 的年轻代中除了拥有 ParNew 收集器是基于并行回收的以外，Parallel 收集器同样也采用了复制算法、并行回收和“Stop-the-World”机制。和 ParNew 收集器不同，Parallel 收集器可以控制程序的吞吐量大小，因此它也被称为吞吐量优先的垃圾收集器。在程序中，开发人员可以通过选项“-XX:GCTimeRatio”设置执行内存回收的时间所占 JVM 运行总时间的比例，也就是控制 GC 的执行频率，公式为 1/(1+N)，默认值为 99，也就是说，将只有 1% 的时间用于执行垃圾回收。除此之外，Parallel 收集器还提供选项“-XX:MaxGCPauseMills”设置执行内存回收时“Stop-the-World”机制的暂停时间阈值，如果指定了该选项，Parallel 收集器将会尽可能地在设定的时间范围内完成内存回收。

需要注意的是，垃圾收集器中吞吐量和低延迟这两个目标本身是相互矛盾的，因为如果选择以吞吐量优先，那么必然需要降低内存回收的执行频率，但是这样会导致 GC 需要更长的暂停时间来执行内存回收。相反的，如果选择以低延迟优先为原则，那么为了降低每次执行内存回收时的暂停时间，也只能频繁地执行内存回收，但这又引起了年轻代内存的缩减和导致程序吞吐量的下降。

举个例子，在 60s 的 JVM 总运行时间里，GC 的执行频率是 20 秒/次，那么 60s 内一共会执行 3 次内存回收，按照每次 GC 耗时 100ms 来计算，最终一共会有 300ms（3×100）被用于执行垃圾回收。但是如果我们将选项“-XX:MaxGCPauseMills”的值调小后，年轻代的内存空间也会自动调整，内存空间越小就越容易被耗尽，也就越容易造成 GC 的执行频繁发生。之前在 60s 的 JVM 总运行时间里，最终会有 300ms 被用于执行内存回收，而如今 GC 的执行频率却是 10s/次，60s 内将会执行 6 次内存回收，按照每次 GC 耗时 60ms 来计算，虽然看上去暂停时间更短了，但最终会耗时 360ms（6×60）用于执行内存回收，很明显程序的吞吐量下降了。所以大家在设置这两个选项时，一定需要注意控制在一个折中的范围之内。Parallel 收集器还提供一个“-XX:UseAdaptiveSizePolicy”选项用于设置 GC 的自动分代大小调节策略，一旦设置这个选项后，就意味着开发人员将不再需要显式地设置年轻代中的一些细节参数，JVM 会根据自身当前的运行情况动态调整这些相关参数。

和 Serial 收集器一样，Parallel 收集器也提供用于执行老年代垃圾收集的 Parallel Old 收集器，Parallel Old 收集器采用了标记-压缩算法，但同样也是基于并行回收和“Stop-the-World”机制。在程序吞吐量优先的应用场景中，Parallel 收集器和 Parallel Old 收集器的组合，在 Server 模式下的内存回收性能很不错。在程序开发过程中，开发人员可以通过选项“-XX:+UseParallelGC”手动指定使用 Parallel 收集器执行内存回收任务。

1. 年轻代并行回收（Parallel Scavenge）收集器

年轻代并行回收收集器也是使用复制算法的收集器。从表面上看，它和并行收集器一样都是多线程、独占式的收集器。但是，并行回收收集器有一个重要的特点，它非常关注系统的吞吐量。

年轻代并行回收收集器可以使用以下选项启用。

- -XX:+UseParallelGC：年轻代使用并行回收收集器，老年代使用串行收集器。
- -XX:+UseParallelOldGC：年轻代和老年代都是用并行回收收集器。

设定线程数量为 24 时，运行输出如代码清单 2-23 所示。

代码清单 2-23　24 个线程的 Parallel 收集器

```
Heap
 PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
  to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
 ParOldGen       total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
  object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
 PSPermGen       total 12288K, used 2054K [0x0f010000, 0x0fc10000,
0x13010000)
  object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

年轻代并行回收收集器可以使用以下选项启用。

- -XX:+MaxGCPauseMills：设置最大垃圾收集停顿时间，它的值是一个大于 0 的整数。收集器在工作时会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。如果希望减少停顿时间，而把这个值设置得很小，为了达到预期的停顿时间，JVM 可能会使用一个较小的堆（一个小堆比一个大堆回收快），而这将导致垃圾回收变得很频繁，从而增加了垃圾回收总时间，减少了吞吐量。
- -XX:+GCTimeRatio：设置吞吐量大小，它的值是一个 0~100 的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n)的时间用于垃圾收集。比如 GCTimeRatio 等于 19，则系统用于垃圾收集的时间不超过 1/(1+19)=5%。默认情况下，它的取值是 99，即不超过 1%的时间用于垃圾收集。

除此之外，并行回收收集器与并行收集器另一个不同之处在于，它支持一种自适应的 GC

调节策略，使用-XX:+UseAdaptiveSizePolicy 可以打开自适应 GC 策略。在这种模式下，年轻代的大小、Eden 和 Survivor 的比例、晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。在手工调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量（GCTimeRatio）和停顿时间（MaxGCPauseMills），让虚拟机自己完成调优工作。

年轻代并行回收收集器的工作日志如代码清单 2-24 所示。

代码清单 2-24　年轻代并行回收收集器工作日志

```
Heap
 PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
  to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
 PSOldGen        total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
  object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
 PSPermGen       total 12288K, used 2054K [0x0f010000, 0x0fc10000,
0x13010000)
  object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

上面的日志显示了回收前的内存大小和回收后的内存大小，以及花费的时间。

2. 老年代并行回收收集器

老年代的并行回收收集器也是一种多线程并发的收集器。和年轻代并行回收收集器一样，它也是一种关注吞吐量的收集器。老年代并行回收收集器使用标记-压缩算法，JDK1.6 之后开始启用。

使用-XX:+UseParallelOldGC 可以在年轻代和老年代都使用并行回收收集器，这是一对非常关注吞吐量的垃圾收集器组合，在对吞吐量敏感的系统中，可以考虑使用。选项-XX:ParallelGCThreads 也可以用于设置垃圾回收时的线程数量。

设置线程数量为 100 时老年代并行回收收集器输出日志如代码清单 2-25 所示。

代码清单 2-25　老年代并行回收收集器线程 100 时日志

```
Heap
 PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
```

```
     to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
    ParOldGen       total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
     object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
    PSPermGen       total 12288K, used 2054K [0x0f010000, 0x0fc10000,
0x13010000)
     object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

2.3.6 CMS 收集器

CMS 全称是 Concurrent-Mark-Sweep。前面说过，在程序吞吐量优先的应用场景中，Parallel 收集器和 Parallel Old 收集器的组合，在 Server 模式下的内存回收性能很不错。但是在某些对系统响应速度要求比较高的项目中，大家总是希望系统能够快速做出响应，而不愿意看到过多的延迟。基于低延迟的考虑，JVM 的设计者们提供了基于并行回收的 CMS（Concurrent-Marking-Sweep）收集器，它是一款优秀的老年代垃圾收集器，也可以被称作 Mostly-Concurrent 收集器。CMS 天生为并发而生，低延迟是它的优势，不过垃圾收集算法却并没有采用标记-复制算法，而是采用标记-清除算法，并且也会因为“Stop-the-world”机制而出现短暂的暂停。

CMS 的执行过程可以分为 4 个主要阶段，即初始标记（Initial-Mark）阶段、并发标记（Concurrent-Mark）阶段、再次标记（Remark）阶段和并发清除（Concurrent-Sweep）阶段。

CMS 收集器的回收周期以一个称为初始标记的阶段开始，在这个阶段中，程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，这个阶段的主要任务就是标记出内存中那些被根对象集合所连接的目标对象是否可达，一旦标记完成之后就会恢复之前被暂停的所有应用线程。接下来将会进入并发标记阶段，而这个阶段的主要任务就是将之前的不可达对象标记为垃圾对象。在 CMS 最终执行内存回收之前，尽管看上去这些垃圾对象都已经被成功标记了，但是由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此在并发标记阶段将无法有效确保之前被标记为垃圾的无用对象的引用关系遭到更改，为了解决这个问题，CMS 会进入到再次标记阶段，这样一来，程序会因为“Stop-the-World”机制而再次出现短暂的暂停，以确保这些垃圾对象都能够被成功且正确地标记。当经历过初始标记、并发标记和再次标记这三个阶段后，CMS 最终将会进入到并发清除阶段执行内存回收，释放掉无用对象所占用的内存空间。

尽管 CMS 收集器采用的是并行回收，但是在其初始化标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程，不过暂停时间并不会太长，因此可以说明目前所有的垃圾收集器都做不到完全不需要“Stop-the-World”，只是尽可能地缩短暂停时间。

之前介绍的几款老年代垃圾收集器，比如 Serial Old 收集器、Parallel Old 收集器的垃圾收集算法都是采用标记-压缩来避免执行 Full GC 后产生内存碎片，而 CMS 收集器的垃圾收集算法采用的是标记-清除算法，这意味着每次执行完内存回收后，由于被执行内存回收的无用对象所占用的内存空间极有可能是不连续的一些内存块，不可避免地将会产生一些内存碎片。那么 CMS 在为新对象分配内存空间后，将无法使用指针碰撞（Bump the Pointer）技术，而只能选择空闲列表（Free List）执行内存分配。

在 HotSpot 中，当垃圾收集器执行完内存回收后，如果内存空间中产生内存碎片，那么只能选择空闲列表作为内存分配算法为新对象分配内存空间。简单来说，会有 JVM 负责维护一个列表，其中所记录的内容就是当前内存空间中可用内存块的坐标，当执行内存分配时，会从列表中定位到一个与新对象所需内存大小一致的连续内存块用于存储生成的对象实例。考虑到内存碎片存在的弊端，CMS 收集器提供选项“-XX:+UseCMS-CompactAtFullCollection”，用于指定在执行完 Full GC 后是否对内存空间进行压缩整理，以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行，所带来的问题就是停顿时间变得更长了，因此 CMS 收集器还提供另外一个选项“-XX:CMSFullGCs-BeforeCompaction”，用于设置在执行多少次 Full GC 后对内存空间进行压缩整理。除了会产生内存碎片外，CMS 收集器还存在一个不容忽视的问题，那就是在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的，那么在并发标记阶段如果产生新的垃圾对象，CMS 将无法对这些垃圾对象进行标记，最终会导致这些新产生的垃圾对象没有被及时回收，从而只能在下一次执行 GC 时释放这些之前未被回收的内存空间。

尽管 Full GC 大多数时候只会发生在老年代垃圾回收阶段，但是实际上 Full GC 的回收范围却不单单仅限于老年代中，从严格意义上来说，Full GC 的回收范围几乎覆盖了整个堆空间，因此 Full GC 将会比 Minor GC 耗费更长的时间来完成垃圾收集。在 HotSpot 中，除了 CMS 收集器之外的任何老年代垃圾收集器在执行内存回收时，都将会执行 Full GC，只有 G1 收集器较为特殊。CMS 收集器提供选项“-XX:CMSInitiatingOccupanyFraction”，用于设置当老年代中的内存使用率达到多少百分比的时候执行内存回收（低版本的 JDK 默认值为 68%，JDK6 及以上版本默认值为 92%），这里的内存回收范围仅限于老年代，而非整个堆空间，因此通过该选项便可以有效降低 Full GC 的执行次数。当然并不是说使用了 CMS 收集器之后，就永远不会再触发 Full GC 了，一旦 CMS 在执行过程中出现“Promotion Failed”或“Concurrent Mode Failure”时，仍然有可能会触发 Full GC 操作。在程序开发过程中，开发人员可以通过选项“-XX:+UseConcMarkSweepGC”来手动指定使用 CMS 收集器执行内存回收任务。

CMS 收集器在其主要的工作阶段虽然没有暴力地彻底暂停应用程序线程，但是由于它和应

用程序线程并发执行，相互抢占 CPU，所以在 CMS 执行期内会对应用程序吞吐量造成一定的影响。CMS 默认启动的线程数是(ParallelGCThreads+3)/4，ParallelGCThreads 是年轻代并行收集器的线程数，也可以通过-XX:ParallelCMSThreads 参数手工设定 CMS 的线程数量。当 CPU 资源比较紧张时，受到 CMS 收集器线程的影响，应用程序的性能在垃圾回收阶段可能会非常糟糕。

由于 CMS 收集器不是独占式的回收器，在 CMS 回收过程中，应用程序仍然在不停地工作。在应用程序工作过程中，又会不断地产生垃圾。这些新生成的垃圾在当前 CMS 回收过程中是无法清除的。同时，因为应用程序没有中断，所以在 CMS 回收过程中，还应该确保应用程序有足够的内存可用。因此，CMS 收集器不会等待堆内存饱和时才进行垃圾回收，而是当堆内存使用率达到某一阈值时，便开始进行回收，以确保应用程序在 CMS 工作过程中依然有足够的空间支持应用程序运行。

这个回收阈值可以使用-XX:CMSInitiatingOccupancyFraction 来指定，默认是 68，即当老年代的空间使用率达到 68%时，会执行一次 CMS 回收。如果应用程序的内存使用率增长很快，在 CMS 的执行过程中，已经出现了内存不足的情况，此时，CMS 回收将会失败，JVM 将启动老年代串行收集器进行垃圾回收。如果这样，应用程序将完全中断，直到垃圾收集完成，这时，应用程序的停顿时间可能很长。因此，根据应用程序的特点，可以对-XX:CMSInitiating-OccupancyFraction 进行调优。如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低 CMS 的触发频率，减少老年代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发老年代串行收集器。

标记-清除算法将会造成大量内存碎片，离散的可用空间无法分配较大的对象。在这种情况下，即使堆内存仍然有较大的剩余空间，也可能会被迫进行一次垃圾回收，以换取一块可用的连续内存，这种现象对系统性能是相当不利的，为了解决这个问题，CMS 收集器还提供了几个用于内存压缩整理的算法。

选项-XX:+UseCMSCompactAtFullCollection 可以使 CMS 在垃圾收集完成后，进行一次内存碎片整理。内存碎片的整理并不是并发进行的。选项-XX:CMSFullGCsBeforeCompaction 可以用于设定进行多少次 CMS 回收后，进行一次内存压缩。

将-XX:CMSInitiatingOccupancyFractio 设置为 100，设置-XX:+UseCMSCompactAtFullCollection，设置-XX:CMSFullGCsBeforeCompaction，日志输出如代码清单 2-26 所示。

代码清单 2-26　线程 100 个时使用 CMS 收集器日志输出

```
   [GC [DefNew: 825K->149K(4928K), 0.0023384 secs][Tenured: 8704K->661K(10944K),
0.0587725 secs] 9017K->661K(15872K), [Perm : 374K->374K(12288K)], 0.0612037 secs]
[Times: user=0.01 sys=0.02, real=0.06 secs]
   Heap
    def new generation   total 4992K, used 179K [0x27010000, 0x27570000, 0x2c560000)
     eden space 4480K,   4% used [0x27010000, 0x2703cda8, 0x27470000)
     from space 512K,   0% used [0x27470000, 0x27470000, 0x274f0000)
     to   space 512K,   0% used [0x274f0000, 0x274f0000, 0x27570000)
    tenured generation   total 10944K, used 8853K [0x2c560000, 0x2d010000,
0x37010000)
      the space 10944K,  80% used [0x2c560000, 0x2ce057c8, 0x2ce05800, 0x2d010000)
    compacting perm gen  total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
      the space 12288K,   3% used [0x37010000, 0x3706db10, 0x3706dc00, 0x37c10000)
       ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
       rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

如果使用选项-XX:+UseConcMarkSweepGC，表示年轻代使用并行收集器，老年代使用 CMS。输出如代码清单 2-27 所示。

代码清单 2-27　使用-XX:+UseConcMarkSweepGC 选项

```
   [GC [ParNew: 8967K->669K(14784K), 0.0040895 secs] 8967K->669K(63936K),
0.0043255 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
   Heap
    par new generation   total 14784K, used 9389K [0x03f50000, 0x04f50000,
0x04f50000)
     eden space 13184K,  66% used [0x03f50000, 0x047d3e58, 0x04c30000)
     from space 1600K,  41% used [0x04dc0000, 0x04e67738, 0x04f50000)
     to   space 1600K,   0% used [0x04c30000, 0x04c30000, 0x04dc0000)
    concurrent mark-sweep generation total 49152K, used 0K [0x04f50000,
0x07f50000, 0x09f50000)
    concurrent-mark-sweep perm gen total 12288K, used 2060K [0x09f50000,
0x0ab50000, 0x0df50000)
```

并行收集器工作时的线程数量可以使用-XX:ParallelGCThreads 选项指定。一般地，最好与 CPU 数量相当，以避免过多的线程数影响垃圾收集性能。在默认情况下，当 CPU 数量小于 8 个，ParallelGCThreads 的值等于 CPU 数量，大于 8 个 ParallelGCThreads 的值等于 3+[5*CPU_Count]/8]。设置为 8 个线程后输出如代码清单 2-28 所示。

代码清单 2-28　并发收集器的线程设置 8 个时的日志输出

```
   [GC [ParNew: 8967K->676K(14784K), 0.0036983 secs] 8967K->676K(63936K),
0.0037662 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
   Heap
    par new generation   total 14784K, used 9395K [0x040e0000, 0x050e0000,
0x050e0000)
     eden space 13184K,  66% used [0x040e0000, 0x04963e58, 0x04dc0000)
     from space 1600K,  42% used [0x04f50000, 0x04ff9100, 0x050e0000)
     to   space 1600K,   0% used [0x04dc0000, 0x04dc0000, 0x04f50000)
    concurrent mark-sweep generation total 49152K, used 0K [0x050e0000,
0x080e0000, 0x0a0e0000)
    concurrent-mark-sweep perm gen total 12288K, used 2060K [0x0a0e0000,
0x0ace0000, 0x0e0e0000)
```

并发收集器的线程设置为 128 个时的日志输出：

```
   [GC [ParNew: 8967K->664K(14784K), 0.0207327 secs] 8967K->664K(63936K),
0.0208077 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
```

并发收集器的线程设置为 640 个时的日志输出：

```
   [GC [ParNew: 8967K->667K(14784K), 0.2323704 secs] 8967K->667K(63936K),
0.2324778 secs] [Times: user=0.34 sys=0.02, real=0.23 secs]
```

并发收集器的线程设置为 1280 个时的日志输出：

```
   Error occurred during initialization of VM
   Too small new size specified
```

2.3.7　Garbage First（G1）GC

这一章节只是简单地介绍了 G1 GC 的设计思路，第 3 章会重点介绍 G1 GC 特有的 GC 选项及使用方式，第 4 章和第 5 章会重点介绍 G1 GC 的内部实现原理、系统架构。

我们常说，每个人做事的方式不同，产生的结果也会不同。G1 GC 采用了一些与 Parallel GC、Serial GC、CMS GC 不同的方式，从而解决了前三个的很多缺陷。例如，G1 GC 切分堆内存为多个区间（Region），从而避免了很多 GC 操作在整个 Java 堆或者整个年代进行，这就好比让你打扫一个房子，如果只打扫一个小房间，那可比打扫一个大房间节省很多时间。

在 JVM 启动时不需要立即指定哪些 Region 属于年轻代，哪些 Region 属于老年代，因为无论年轻代或老年代，它们都不需要一大块连续的内存块，只是由一系列 Region 组成而已。随着时间的流逝，G1 Region 的映射关系（属于谁）是来来回回地变动的，这印证了一句话，“三

十年河东，三十年河西[1]。”例如开始的时候，Region A 被分配给了年轻代，一个年轻代回收结束后，这个 Region 又被放回了空闲/可用 Region 队列，可能下一次就被分配给了一个老年代对象使用。

G1 的年轻代收集阶段是一个并行的独占收集器。和其他 HotSpot 垃圾收集器一样，当一个年轻代收集进行时，整个年轻代会被回收，所有的应用程序线程会被中断，G1 GC 会启用多线程执行年轻代回收。和年轻代不同，老年代的 G1 回收器和其他的 HotSpot 不同，G1 的老年代回收器不需要整个老年代被回收，一次只需要扫描/回收一小部分老年代的 Region 就可以了。此外，需要注意的是，这个老年代 Region 是和年轻代一起被回收的。

和 CMS GC 类似，当老年代空间耗尽时，G1 GC 会启动一个失败保护[2]的应急机制，该机制会收集、压缩整个老年代。

G1 很重视老年代的垃圾回收，一旦整个堆空间占有率达到指定的阈值（启动时可配置），G1 会立即启动一个独占的并行初始标记阶段（initial-mark phase）进行垃圾回收。在 CMS GC 里，我们只需要单独判断老年代的占有率，而在 G1 GC，判断的是整个 Java 堆内部老年代的占有率，就是前面所说的堆占有率，足以见 G1 对老年代的重视[3]。

初始标记阶段一般和年轻代 GC 一起运行，一旦初始标记阶段结束，并行多线程的标记阶段就开始启动去标记所有老年代还存活的对象，注意这个标记阶段不是独占式的，它允许应用程序线程和它并行执行。当这个标记阶段运行完毕之后，为了再次确认是否有逃过扫描的对象，[4]启动一个独占式的再次标记阶段（remark phase），尝试标记所有遗漏的对象。在这个再次标记阶段结束之后，G1 就掌握了所有的老年代 Region 的标记信息，这和国家的户口统计方式差不多。一旦老年代的某些 Region 内部存在任何的存活对象，它就可以在下一个阶段，即清除阶段（cleanup phase）被清除了，就是可以销户了，又被放回了可用 Region 队列[5]。同样地，再次标记阶段结束后就可以对一些老年代执行收集动作。

前面提到了 CSet 概念，一个 CSet 里面可以包含多少 Region 取决于多少空间可以被释放、G1 停顿目标时间这两个因素。前面说起过混合 GC（Mixed GC），这里就要具体说明一下了。当 CSet 被确定之后，会在接下来的一个年轻代回收过程当中对 CSet 进行回收，通过年轻代 GC

1　一句广泛流传的谚语，用来形容世事盛衰兴替，感叹世事变化无常。较早的引用出现在清吴敬梓《儒林外史》。

2　即 Fail Safe，在几乎所有的硬件产品、软件产品中都存在失败保护机制，以解决一些异常问题导致的极端情况。

3　这和国家的政策一样，对老年人很重视。

4　这是因为毕竟是和应用程序线程并行执行的，也许你扫描过后应用程序又新增了对象。

5　可以有多种说法，Available Region、Unused Region、Free Region 等。

的几个阶段，一部分的老年代 Region 会被回收并放入年轻代使用。这个概念很灵活，即 G1 只关注你有没有存活对象了，如果没有，无论你属于老年代，还是属于年轻代，你都会被回收并放入可用 Region 队列，下一次你被分配到哪里就不确定了。也正是因为 Region、混合收集这些特性，让 G1 对老年代的垃圾收集方式有别于 Serial GC、Parallel GC 和 CMS GC，G1 采用 Region 方式让对象之间的联系存在于虚拟地址之上，这样就不需要针对老年代的压缩和回收动作对整个 Java 堆执行扫描，为老年代回收节约了时间。

由于 G1 是基于 Region 的 GC，所以它适用于大内存的机器，这样就可以分配大量内存给 Java 堆内存，即使很大，也仅是对 Region 进行扫描，性能还是很高的。

G1 的一个最大的贡献是它可以让我们设置最大停顿时间，只要设置了这个时间，G1 就会通过自动调整年轻代空间大小和整体 Java 堆空间大小来匹配这个目标停顿时间。比如你设置了一个很短的停顿时间，G1 会设置比较小的年轻代、比较大的整个 Java 堆空间，对应的老年代也会比较大[1]。

总的来说，G1 和 CMS 的目标是构建针对大内存回收的较短停顿时间，如果想要提高应用的内部吞吐量(虽然这也是 G1 的目标)，也可以忍受较长时间的停顿时间，那么还是选择 Parallel GC 吧，毕竟它是专门为高吞吐量而研发的。

1. G1 设计思路

G1 把整个 Java 堆划分为若干个区间（Region）。每个 Region 大小为 2 的倍数，范围在 1MB~32MB 之间，可能为 1MB、2MB、4MB、8MB、16MB、32MB。所有的 Region 有一样的大小，在 JVM 生命周期内不会被改变。例如，我们通过命令行启动 JVM 进程，配置选项-Xmx16g –Xms16g，即设置 16GB 的堆大小，假设也配置了 2000 个 Region，则每个 Region=16GB/2000= 8MB。如果堆的大小很大，而每个 Region 的大小很小，则 Region 数量可能会超过 2000。同样，很小的堆大小会导致 Region 数量很少。每个 Region 同时只会被用于一个用途，例如年轻代对象。

前面讲过，CSet 包含了一系列的 Region，相应地，每个 Region 有一个 RSet[2]概念。这个 RSet 包含了相应 Region 里面所有指针的位置集合。RSet 的大小和 Region 数量有直接关系，一般来说 RSet 的大小占整个 Java 堆空间的 1%~20%。

G1 包括了几个类型的 Region。可用 Region 就是 Unused Region，Eden Region 和 Survivor

---

[1] 这是因为年轻代回收很频繁，你会在后面给出的示例 GC 输出里看到大量的年轻代回收。

[2] 全称 Remembered Set。

Region 组成了年轻代空间；相应地，Eden Region 组成了年轻代的 Eden 空间；Survivor（幸存者）Region 组成了年轻代的 Eden 空间。注意，在年轻代、混合代、Full GC 这三个阶段，年轻代的 Eden Region 和 Survivor Region 的数量会随时变化。Humongous Region（大对象 Region）是老年代 Region 的一部分，里面的对象超过每个 Region 的 50%空间，这一点有别于一般对象 Region。

从之前的介绍我们知道没有必要去刻意区分 Region 的用途，因为 G1 设计 Region 的分配原则是很灵活的。一开始 G1 会从可用 Region 队列里面挑选出 Region 并设置为 Eden Region，一个 Eden Region 里面填满对象以后，又会从可用 Region 队列里再挑出一个。当所有的 Eden Region 都被填满时，一个年轻代 GC 收集就会开始执行了，在这个收集阶段，我们会收集 Eden 和 Survivor Region，所有的存活对象要么进入到下一个 Survivor Region，要么进入老年代 Region。

G1 提供了一个选项-XX:InitiatingHeapOccupancyPercent，默认值是 Java 堆空间的 45%，这个选项决定了是否开始一次老年代回收动作，即年轻代 GC 结束之后，G1 会评估剩余的对象是否达到了 45%这个阈值。

如果标记阶段（Marking Phase）结束后一个老年代的 Region 已经不存在对象，那么它会被放回可用 Region 队列，反之，它会被放入混合收集器。这些知识会在后续章节详细介绍。

由于标记阶段不是一个独占式的多线程并行程序，这样应用程序线程就会和它一起并行执行。为了避免标记阶段占用过多的 CPU 资源，G1 采用时间片方式分段执行操作，即在时间片内全力运行，然后休息一段时间，这个休息时间就是让应用程序尽可能多地使用 CPU 资源运行。

2. 大对象（Humongous Object）

G1 对于大对象有特殊的分配方式。前面介绍过，一个大对象是指该对象的大小超过一个 Region 大小的 50%以上。这个大小包括了 Java 对象头。

打个比方，一大家子人在北京等待分配房子，要分在一个四合院里面，房间之间要相邻，这样方便互相照顾、串门。G1 采用的方式也是类似的，G1 会挑选出一组连续的可用 Region，相加后只要能够确保总大小可以存放这个大对象，就会分配给这个大对象。反之，如果没有能够找到符合条件的连续可用 Region，那么 G1 只能执行一次 Full GC，用于压缩 Java 堆。这片可用的连续 Region 由大对象开始（Humongous Start）Region 和大对象持续（Humongous Continue）Region 组成。

大对象 Region 属于老年代的一部分，它只包含一个对象。当并行标记阶段发现没有存活对象时，G1 会回收这个大对象 Region，注意这个动作可以是一个批量回收。

注意，大对象在以下场景会引起性能问题。

- 大对象的存活周期很短。
- 满足第 1 条时会在年轻代回收对象 Region。
- 频繁地分配大对象。

3. 全垃圾收集（Full Garbage Collection）

G1 的 Full GC 和 Serial GC 的 Full GC 采用的是同一种算法。Full GC 会对整个 Java 堆进行压缩。G1 的 Full GC 是单线程的，会引起较长的停顿时间，因此 G1 的设计目标是减少 Full GC 的发生次数。

4. 并行循环（Concurrent Cycle）

一个 G1 并行循环包括几个阶段的活动：初始标记（Initial Marking）、并行 Root 区间扫描（Concurrent Root Region Scanning）、并行标记（Concurrent Marking）、重标记（Remarking）和清除（Cleanup）。除了最后的 Cleanup 阶段以外，其余阶段都属于标记存活对象阶段。

初始标记阶段的目的是收集所有 GC 根（Roots）。Roots 是一个对象的起源指针。为了收集根引用，从应用线程开始，应用线程必须停下来，所以初始标记阶段是一个独占式的。由于一个年轻代 GC 必须收集所有的 Roots，所以 G1 的初始标记在一个年轻代 GC 里完成。

并行根区间扫描阶段必须扫描和标记所有幸存者区间的对象引用，这一阶段所有的应用程序线程都可以并行执行，唯一的约束是扫描必须在下一个 GC 开始前完成。这一约束的原因是一个新的 GC 事件会产生一堆新的幸存者对象集合，这些对象和初始化标记阶段的幸存者对象不一样，容易发生混淆。

并行标记阶段完成了几乎所有的标记工作。在这一阶段，利用多线程并行标记存活对象及对应的逻辑地图。这一阶段允许所有的 Java 线程并行执行，但是对应用程序来说总体的吞吐量可能会下降。其实任何一个系统都和人体循环一样，当没有外部干扰时，系统可以正常运行，如果受到外部干扰，人体系统也会出现混乱，甚至出现短时间的休克。

重标记阶段是一个独占式阶段，通常是一个很短的停顿，这个阶段会完成所有的标记工作。

最后一个并行标记步骤是清除阶段。在这个阶段，没有包含存活对象的 Region 会被回收，并随即被加入可用 Region 队列。这个阶段的重要意义是最终决定了哪些 Region 可以进入混合 GC。在 G1 内部，混合 GC 是非常重要的释放内存机制，避免了 G1 出现没有可用 Region 的情况发生，否则就会触发 Full GC 事件。

5. 堆大小（Heap Sizing）

G1 的 Java 堆通常由多个 Region 组成，前面说过，最多可能会有 2000 个 Region 存在。G1 设置堆大小的选项和其他 GC 一样，都是由 –xms 和 –xmx 配置。

G1 在以下几种情况下可能会增大堆内存大小。

- Full GC 阶段。
- Young 或 Mixed GC 发生时，G1 计算 GC 花费的时间与 Java 线程的花费时间比例，如果-XX:GCTimeRatio 设置 GC 花费时间很长，则堆大小会增大，这样的设计思路是希望 G1 发生 GC 的频率降低，这样 GC 花费时间和 Java 线程花费时间比例也会相应下降。

  -XX:GCTimeRatio 选项的默认值是 9，所有其他 HotSpot GC 的默认值是 99。这个值越大，代表 Java 堆空间大小增长越偏激，即越容易扩大堆空间大小，这样也是为了达到降低 GC 花费时间的设计目标。

- 如果一个对象分配失败，即便一个 GC 刚刚结束，G1 采用的策略不是立即重复 Full GC，而是通过增大堆内存大小，确保对象分配成功。这样的设计理念符合 G1 的避免 Full GC 发生的最初思想。
- 和第 3 条一样，如果出现一个大对象分配失败，前面说过，大对象需要几个连续的 Region 区间才能确保对象分配成功。如果发生这种分配失败的情况，采用的设计理念也不是调用 Full GC，而是扩大堆内存。
- 当 GC 申请加入一个新的 Region 时。

引用一段在 StackOverfall.com 上看到的经验分享，“我在一个真实的、较大规模的应用程序中使用过 G1：大约分配有 60GB~70GB 内存，存活对象大约在 20GB~50GB 之间。服务器运行 Linux 操作系统，JDK 版本为 6u22。G1 与 PS/PS Old 相比，最大的好处是停顿时间更加可控、可预测。如果我在 PS 中设置一个很低的最大允许 GC 时间，譬如期望 50ms 内完成 GC（-XX:MaxGCPauseMillis=50），但在 65GB 的 Java 堆下有可能得到的直接结果是一次长达 30s 至 2min 的漫长的 Stop-the-World 过程。而 G1 与 CMS 相比，它们都立足于低停顿时间，CMS 仍然是我现在的选择，但是随着 Oracle 对 G1 的持续改进，我相信 G1 会是最终的胜利者。如果你现在采用的收集器没有出现问题，那么就没有任何理由现在去选择 G1；如果你的应用追求低停顿，那么 G1 现在已经可以作为一个可尝试的选择；如果你的应用追求吞吐量，那么 G1 并不会为你带来什么特别的好处。”

2.4 常见问题解析

2.4.1 jmap -heap 或-histo 不能用

假设遇到这样一个场景，登录某台机器（假设是 Linux 操作系统，需要通过 SSH 工具登录）并执行 jstack 命令，报错“get_thread_regs failed for a lwp”。既然 jstack 没法做，就 pstack 看看进程到底什么状况吧，于是 pstack [pid]进一步调查，发现有一个线程的堆栈信息有点奇怪：“\#0 0x00000038e720cd91 in sem_wait ()”。这个错误是由于进程在等信号，这个时候通常会阻塞其他所有的线程，于是立刻 PS 看了下进程的状态，如果进程的状态变成了 T，那么上面碰到的所有现象都很容易解释了，于是执行 kill -CONT [pid]，一切都会恢复正常。

问题虽然解决了，但是要找深层次的原因。有一种原因，在使用 CMS GC 的情况下，执行 jmap -heap 有些时候会导致进程变 T，因此强烈建议别执行这个命令，如果想获取内存目前每个区域的使用状况，可通过 jstat -gc 或 jstat -gccapacity 来拿到。

2.4.2 YGC 越来越慢

假设遇到的现象是 YGC 越来越慢，但可以肯定不是由于 CMS GC 碎片问题造成的。假设运行程序如代码清单 2-29 实验程序所示。

代码清单 2-29 实验程序

```
import com.thoughtworks.xstream.XStream;

public class XStreamTest {

    public static void main(String[] args) throws Exception {
        while(true){
            XStream xs = new XStream();
            xs.toString();
            xs = null;
        }
    }

}
```

设置 JVM 选项为“-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xms512m -Xmx512m -Xmn100m -XX:+UseConcMarkSweepGC”。应该就可以看到 ygc 的速度从 10ms 多一直增长到 100ms 多。在解决很多问题的时候，工具起的作用往往巨大，很多时候通过工具分析，很快便

能找到原因，但是这次并没有，图 2-7 是 VisualVM 观察到堆上的 GC 图表。

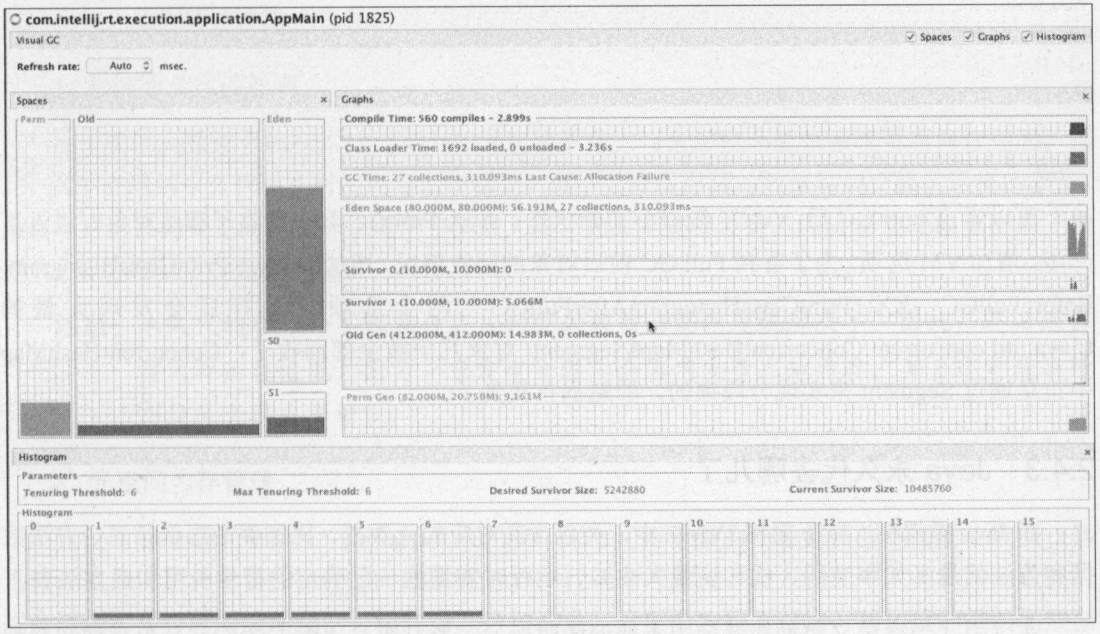


图 2-7　堆上 GC 状态图

从图中可以看出，Perm 区空间基本水平，但是 Old 区空间呈增长态势与 YGCT 时间增长的倍率基本一致。熟悉 YGCT 的朋友都知道 YGC 主要分为标记和收回两个阶段，YGCT 也是基于这两个阶段统计的。由于每次回收的空间大小差不多，所以怀疑是标记阶段使用的时间比较多。

基于对 GC Roots 的怀疑，猜测 Old 区中存在越来越多的 GC Root 节点。为了进一步验证是 Old 区的 GC Roots 造成 YGCT 增加的，我们来做一次 Full GC，去掉 Old 区。如代码清单 2-30 所示。

代码清单 2-30　实验程序

```
public class TestSlowYGC {
public static void main(String[] args) throws Exception {
int i= 0;
while (true) {
XStream xs = new XStream();
xs.toString();
xs = null;
```

```
    if(i++ % 10000 == 0)
    {
    System.gc();
    }
    }
    }
    }
```

可以看出 Full GC 后 YGCT 锐减到初始状态。那么 Full GC 到底回收了哪些对象？进入下一步，增加 VM 选项。为了看到 Full GC 前后对象的回收情况，增加“-XX:+ PrintClassHistogram-BeforeFullGC -XX:+PrintClassHistogramAfterFullGC ”两个选项。运行后会发现大量的 Ljava.util.concurrent.ConcurrentHashMap$Segment 对象几乎被全部回收了，ConcurrentHashMap 中正是通过 Segment 来实现分段锁的，罪魁祸首找到了。

2.4.3 Java 永久代去哪儿了

在 Java 虚拟机（以下简称 JVM）中，类包含其对应的元数据，比如类的层级信息、方法数据和方法信息（如字节码、栈和变量大小）、运行时常量池、已确定的符号引用和虚方法表。

在过去（当自定义类加载器使用不普遍的时候），类几乎是“静态的”并且很少被卸载和回收，因此类也可以被看成“永久的”。另外由于类作为 JVM 实现的一部分，它们不由程序来创建，因为它们也被认为是“非堆”的内存。

在 JDK8 之前的 HotSpot 虚拟机中，类的这些“永久的”数据存放在一个叫作永久代的区域。在 JVM 启动之前可以通过设置-XX:MaxPermSize 的值来控制永久代的大小，32 位机器默认的永久代的大小为 64MB，64 位的机器则为 85MB。永久代的垃圾回收和老年代的垃圾回收是绑定的，一旦其中一个区域被占满，这两个区都要进行垃圾回收。但是有一个明显的问题，由于可以通过-XX:MaxPermSize 设置永久代的大小，一旦类的元数据超过了设定的大小，程序就会耗尽内存，并出现内存溢出错误（OOM）。

随着 Java8 的到来，再也见不到永久代了。但是这并不意味着类的元数据信息也消失了。这些数据被移到了一个与堆不相连的本地内存区域，这个区域就是我们要提到的元空间。

这项改动是很有必要的，因为对永久代进行调优是很困难的。永久代中的元数据可能会随着每一次 Full GC 发生而进行移动，而且为永久代设置空间大小也是很难确定的，因为这其中有很多影响因素，比如类的总数、常量池的大小和方法数量等。

同时，HotSpot 虚拟机的每种类型的垃圾回收器都需要特殊处理永久代中的元数据。将元数

据从永久代剥离出来，不仅实现了对元空间的无缝管理，还可以简化 Full GC 以及对以后的并发隔离类元数据等方面进行优化。

1. 移除永久代的影响

由于类的元数据分配在本地内存中，元空间的最大可分配空间就是系统可用内存空间。因此，就不会遇到永久代存在时的内存溢出错误，也不会出现泄漏的数据移到交换区这样的事情。最终用户可以为元空间设置一个可用空间最大值，如果不进行设置，JVM 会自动根据类的元数据大小动态增加元空间的容量。

注意： 永久代的移除并不代表自定义的类加载器泄漏问题就解决了。因此，还必须监控内存消耗情况，因为一旦发生泄漏，会占用大量本地内存，并且还可能导致交换区交换更加糟糕。

2. 元空间内存管理

元空间的内存管理由元空间虚拟机来完成。先前，对于类的元数据需要不同的垃圾回收器进行处理，现在只需要执行元空间虚拟机的 C++代码即可完成。在元空间中，类和其元数据的生命周期和其对应的类加载器是相同的。换句话说，只要类加载器存活，其加载的类的元数据也是存活的，因而不会被回收掉。

从行文到现在提到的元空间稍微有点不严谨。准确地说，每一个类加载器的存储区域都称作一个元空间，所有的元空间合在一起就是我们一直说的元空间。当一个类加载器被垃圾回收器标记为不再存活时，其对应的元空间会被回收。在元空间的回收过程中没有重定位和压缩等操作，但是元空间内的元数据会进行扫描来确定 Java 引用。

元空间虚拟机负责元空间的分配，其采用的形式为组块分配。组块的大小因类加载器的类型而异。在元空间虚拟机中存在一个全局的空闲组块列表。当一个类加载器需要组块时，它就会从这个全局的组块列表中获取并维持一个自己的组块列表。当一个类加载器不再存活，其持有的组块将会被释放，并返回给全局组块列表。类加载器持有的组块又会被分成多个块，每一个块存储一个单元的元信息。组块中的块是线性分配的（指针碰撞分配形式），组块分配自内存映射区域。这些全局的虚拟内存映射区域以链表形式连接，一旦某个虚拟内存映射区域清空，这部分内存就会返回给操作系统。

3. 元空间调优与工具

正如上面提到的，元空间虚拟机控制元空间的增长。但是有些时候我们想限制其增长，比如通过显式在命令行中设置-XX:MaxMetaspaceSize。默认情况下，-XX:MaxMetaspaceSize 的值

没有限制，因此元空间甚至可以延伸到交换区，但是这时候进行本地内存分配会失败。

对于一个 64 位的服务器端 JVM 来说，其默认的 - XX:MetaspaceSize 值为 21MB。这就是初始的高水位线，一旦触及这个水位线，Full GC 将会被触发并卸载没用的类（即这些类对应的类加载器不再存活），然后这个高水位线将会重置。新的高水位线的值取决于 GC 后释放了多少元空间。如果释放的空间不足，则这个高水位线上升。如果释放空间过多，则高水位线下降。如果初始化的高水位线设置过低，上述高水位线调整情况会发生很多次。通过垃圾回收器的日志可以观察到 Full GC 多次调用。为了避免频繁地 GC，建议将 - XX:MetaspaceSize 设置为一个相对较高的值。

经过多次 GC 之后，元空间虚拟机自动调节高水位线，以此来推迟下一次垃圾回收的到来。

有这样两个选项：-XX:MinMetaspaceFreeRatio 和-XX:MaxMetaspaceFreeRatio，它们类似于 GC 的 FreeRatio 选项，用来设置元空间空闲比例的最大值和最小值。可以通过命令行设置这两个选项的值。

下面是一些改进的工具，用来获取更多关于元空间的信息。

- jmap -clstats PID：打印类加载器数据（-clstats 是-permstat 的替代方案，在 JDK8 之前，-permstat 用来打印类加载器的数据）。
- jstat -gc LVMID：用来打印元空间的信息。
- jcmd PID GC.class_stats：一个新的诊断命令，用来连接到运行的 JVM 并输出详尽的类元数据的柱状图。

前面已经提到，元空间虚拟机采用了组块分配的形式，同时区块的大小由类加载器类型决定。类信息并不是固定大小的，因此有可能分配的空闲区块和类需要的区块大小不同，这种情况下可能导致碎片存在。元空间虚拟机目前并不支持压缩操作，所以碎片化是目前最大的问题。

2.5 本章小结

本章首先对 Java 虚拟机内存模型进行了阐述，逐一介绍了堆内存、栈、方法区等，随后开始介绍垃圾收集算法，针对每一种算法逐一解释实现原理，接下来逐一介绍了各代 JDK 对应的 GC，以及每款 GC 的特性，通过一些实例让读者能理解其工作原理。最后对一些常见问题进行了讲解。下一章会重点介绍 G1 GC 的特有选项（参数），也会附带介绍其他 GC 的运行输出，这些是为了让读者能够更好地对比输出。

3

第 3 章

G1 GC 应用示例

美剧《实习医生格蕾》是一部非常棒的医学故事类电视剧。在第十季中男主角德立克不幸遇到了车祸，在自己被医生进行急救处理时依然保持着外科医生的超高职业素养，试图和医生说需要先扫描脑电图，否则不能排除自己脑部存在积血的可能性，因为这个原因对于人类来说可能是致命性的。他的这个职业反应和我们程序员使用 G1 GC 时的过程是一样的，当我们的个人能力不够强大时，我们需要按照专家给出的诊断选项逐一操作，这样能够快速定位出现问题的原因并着手解决[1]。

G1 GC 给我们提供了很多的命令行选项，也就是参数，这些参数一类以布尔类型打头，“+”表示启用该选项，“-”表示关闭该选项。另一类采用数字赋值，不需要布尔类型打头。我们会在本章具体讲解每一个常用选项的意义，并且用一个实际程序作为示例。通过这样的方式让读者为第 4 章、第 5 章做好准备。

---

[1] Trouble Shooting（故障排除）的能力很难培养，需要个人的逻辑能力、技术能力，以及持之以恒的积累。

注意，本章示例的选项基于 JDK8U45 运行。

本章主要介绍和解决以下问题，这些也是后续章节的实践部分。

- 确定一个简单的代码程序作为示例。
- 了解 G1 GC 自带的各种命令行选项。
- 了解各个选项单独或混合使用后的运行输出。
- 对 G1 GC 输出日志进行一些解释。

3.1 范例程序

本章会介绍几十个 G1 GC 的常用选项（中国学生一般喜欢用参数这个词，我觉得既然是命令行的配置选项，还是用选项来标识比较形象），其中有几个选项会和第 2 章介绍的内容重复，这是为了让读者更好地了解 G1 GC，我的思路是不断重复，这样印象会更深刻，希望读者理解。

我们需要一个程序可以支撑起我们的选项使用、日志输出解释等需求，所以这里选择了一个程序作为选项的统一运行载体，如果没有特别注明，本章的选项运行使用的就是这个程序。先问一句，GC 什么时候触发？一定是在堆内存分配失败的时候触发，这个就是有需求，自然也就有对策。

代码清单 3-1 所示的程序中，定义了一个叫作 GreenhouseScheduler 的类，这个类引用了多线程包和工具包，这两个包都是 JDK 自带的库程序。这是一个简单的任务调度实现类，包含了 schedule、repeat 等方法，也包含了 LightOn、LightOff、WaterOn、WaterOff、CollectData 等内部类，最后在 Main 函数里逐一调用各个类并启动线程，通过 repeat 方法反复执行，打印输出如代码清单 3-2 所示。

代码清单 3-1　范例程序

```
 import Java.util.concurrent.*;
import Java.util.*;

public class GreenhouseScheduler {
   private volatile boolean light = false;
   private volatile boolean water = false;
   private String thermostat = "Day";

   public synchronized String getThermostat() {
      return thermostat;
```

```
}

public synchronized void setThermostat(String value) {
   thermostat = value;
}

ScheduledThreadPoolExecutor scheduler = new ScheduledThreadPoolExecutor(10);

public void schedule(Runnable event, long delay) {
   scheduler.schedule(event,delay,TimeUnit.MILLISECONDS);
}

public void repeat(Runnable event, long initialDelay, long period) {
  scheduler.scheduleAtFixedRate(
    event, initialDelay, period, TimeUnit.MILLISECONDS);
}

class LightOn implements Runnable {
   public void run() {
     //硬件控制代码，用于打开灯
     System.out.println("Turning on lights");
     light = true;
   }
}

class LightOff implements Runnable {
   public void run() {
     // 硬件控制代码，用于关闭灯
     System.out.println("Turning off lights");
     light = false;
   }
 }

class WaterOn implements Runnable {
   public void run() {
     // 硬件控制代码
     System.out.println("Turning greenhouse water on");
     water = true;
   }
}
```

```
    class WaterOff implements Runnable {
      public void run() {
        // 硬件控制代码
        System.out.println("Turning greenhouse water off");
        water = false;
      }
    }

    class ThermostatNight implements Runnable {
      public void run() {
        // 硬件控制代码
        System.out.println("Thermostat to night setting");
        setThermostat("Night");
      }
    }

    class ThermostatDay implements Runnable {
      public void run() {
        // 硬件控制代码
        System.out.println("Thermostat to day setting");
        setThermostat("Day");
      }
    }

    class Bell implements Runnable {
      public void run() { System.out.println("Bing!"); }
    }

    class Terminate implements Runnable {
      public void run() {
        System.out.println("Terminating");
        scheduler.shutdownNow();
        // 必须启动一个独立的任务，因为调度服务已经关闭
        new Thread() {
          public void run() {
            for(DataPoint d : data)
            System.out.println(d);
          }
      }.start();
    }
  }
```

```
    // 新的特征数据收集
    static class DataPoint {
      final Calendar time;
      final float temperature;
      final float humidity;

      public DataPoint(Calendar d, float temp, float hum) {
        time = d;
        temperature = temp;
        humidity = hum;
      }

      public String toString() {
        return time.getTime() +
          String.format(
  " temperature: %1$.1f humidity: %2$.2f",
          temperature, humidity);
        }
      }

      private Calendar lastTime = Calendar.getInstance();
      { // 调试时间为半小时
        lastTime.set(Calendar.MINUTE, 30);
        lastTime.set(Calendar.SECOND, 00);
      }

      private float lastTemp = 65.0f;
      private int tempDirection = +1;
      private float lastHumidity = 50.0f;
      private int humidityDirection = +1;
      private Random rand = new Random(47);
      List<DataPoint> data = Collections.synchronizedList(new
ArrayList<DataPoint>());

      class CollectData implements Runnable {
        public void run() {
        System.out.println("Collecting data");
        synchronized(GreenhouseScheduler.this) {
          // 假装时间间隔比它长:
          lastTime.set(Calendar.MINUTE,lastTime.get(Calendar.MINUTE) + 30);
         // 1/5 的机会转变方向:
```

```
            if(rand.nextInt(5) == 4)
              tempDirection = -tempDirection;
            // 存储前一个值:
              lastTemp = lastTemp + tempDirection * (1.0f + rand.nextFloat());
            if(rand.nextInt(5) == 4)
              humidityDirection = -humidityDirection;
              lastHumidity = lastHumidity + humidityDirection * rand.nextFloat();
            // 日历可以复制，否则所有的数据点保持
            // 上一时间段的属性。对于类似于日历这
            // 样的对象，clone()方法就可以了。
            data.add(new DataPoint((Calendar)lastTime.clone(),lastTemp,
lastHumidity));
          }
        }
    }

    public static void main(String[] args) {
        GreenhouseScheduler gh = new GreenhouseScheduler();
        gh.schedule(gh.new Terminate(), 5000);
        // 不需要“Restart”类:
        gh.repeat(gh.new Bell(), 0, 1000);
        gh.repeat(gh.new ThermostatNight(), 0, 2000);
        gh.repeat(gh.new LightOn(), 0, 200);
        gh.repeat(gh.new LightOff(), 0, 400);
        gh.repeat(gh.new WaterOn(), 0, 600);
        gh.repeat(gh.new WaterOff(), 0, 800);
        gh.repeat(gh.new ThermostatDay(), 0, 1400);
        gh.repeat(gh.new CollectData(), 500, 500);
    }
}
```

代码清单 3-2　范例程序 3-1 运行输出

```
Bing!
Thermostat to night setting
Turning on lights
Turning off lights
Turning greenhouse water on
Turning greenhouse water off
Thermostat to day setting
Turning on lights
Turning on lights
```

```
Turning off lights
Collecting data
Turning on lights
Turning greenhouse water on
Turning on lights
Turning off lights
Turning greenhouse water off
Bing!
Turning on lights
Collecting data
Turning on lights
Turning off lights
Turning greenhouse water on
Turning on lights
Thermostat to day setting
Collecting data
Turning on lights
Turning off lights
Turning greenhouse water off
Turning on lights
Turning greenhouse water on
Bing!
Thermostat to night setting
Turning on lights
Turning off lights
Collecting data
Turning on lights
Turning on lights
Turning off lights
Turning greenhouse water on
Turning greenhouse water off
Collecting data
Turning on lights
Turning on lights
Turning off lights
Thermostat to day setting
Bing!
Turning on lights
Turning greenhouse water on
Collecting data
Turning off lights
```

```
Turning on lights
Turning greenhouse water off
Turning on lights
Collecting data
Turning on lights
Turning off lights
Turning greenhouse water on
Turning on lights
Bing!
Thermostat to night setting
Turning on lights
Turning off lights
Turning greenhouse water off
Collecting data
Turning greenhouse water on
Thermostat to day setting
Turning on lights
Turning on lights
Turning off lights
Collecting data
Turning on lights
Turning on lights
Turning off lights
Turning greenhouse water on
Turning greenhouse water off
Terminating
Mon Aug 01 23:00:00 CST 2016 temperature: 66.4 humidity: 50.05
Mon Aug 01 23:30:00 CST 2016 temperature: 68.0 humidity: 50.47
Tue Aug 02 00:00:00 CST 2016 temperature: 69.7 humidity: 51.42
Tue Aug 02 00:30:00 CST 2016 temperature: 70.8 humidity: 50.87
Tue Aug 02 01:00:00 CST 2016 temperature: 72.0 humidity: 50.32
Tue Aug 02 01:30:00 CST 2016 temperature: 73.2 humidity: 49.92
Tue Aug 02 02:00:00 CST 2016 temperature: 71.9 humidity: 49.81
Tue Aug 02 02:30:00 CST 2016 temperature: 70.1 humidity: 50.25
Tue Aug 02 03:00:00 CST 2016 temperature: 68.9 humidity: 51.00
```

3.2 选项解释及应用

首先在 cmd 命令行模式下输入 Java-X，如 C:\Users\Administrator>Java –X。输出如代码清

单 3-3 所示，从输出可以看到 GC 日志可以通过-Xloggc:<file>方式输出，这个在后续的详细描述里面会逐一讲解。

代码清单 3-3　Java –X 运行输出

```
    -Xmixed                  混合模式执行 (默认)
    -Xint                    仅解释模式执行
    -Xbootclasspath:         <用 ; 分隔的目录和 zip/jar 文件>
                             设置搜索路径以引导类和资源
    -Xbootclasspath/a:       <用 ; 分隔的目录和 zip/jar 文件>
                             附加在引导类路径末尾
    -Xbootclasspath/p:       <用 ; 分隔的目录和 zip/jar 文件>
                             置于引导类路径之前
    -Xdiag                   显示附加诊断消息
    -Xnoclassgc              禁用类垃圾收集
    -Xincgc                  启用增量垃圾收集
    -Xloggc:<file>           将 GC 状态记录在文件中 (带时间戳)
    -Xbatch                  禁用后台编译
    -Xms<size>               设置初始 Java 堆大小
    -Xmx<size>               设置最大 Java 堆大小
    -Xss<size>               设置 Java 线程堆栈大小
    -Xprof                   输出 CPU 配置文件数据
    -Xfuture                 启用最严格的检查, 预期将来的默认值
    -Xrs                     减少 Java/VM 对操作系统信号的使用 (请参阅文档)
    -Xcheck:jni              对 JNI 函数执行其他检查
    -Xshare:off              不尝试使用共享类数据
    -Xshare:auto             在可能的情况下使用共享类数据 (默认)
    -Xshare:on               要求使用共享类数据, 否则将失败
    -XshowSettings           显示所有设置并继续
    -XshowSettings:all       显示所有设置并继续
    -XshowSettings:vm        显示所有与 vm 相关的设置并继续
    -XshowSettings:properties    显示所有属性设置并继续
    -XshowSettings:locale        显示所有与区域设置相关的设置并继续
```

1. –XX:+PrintGCDetails

该选项用于记录 GC 运行时的详细数据信息并输出，是最基本、使用最普遍的一个选项。这个选项适用于所有 GC，输出内容主要包括新生成对象占用内存大小以及耗费时间、各个年龄代的情况、每次回收的对应数据等。如果不指定 GC（还是采用 JDK8U45），来看一看 GC 运行输出，可以通过这个选项进行输出，如图 3-1 所示。

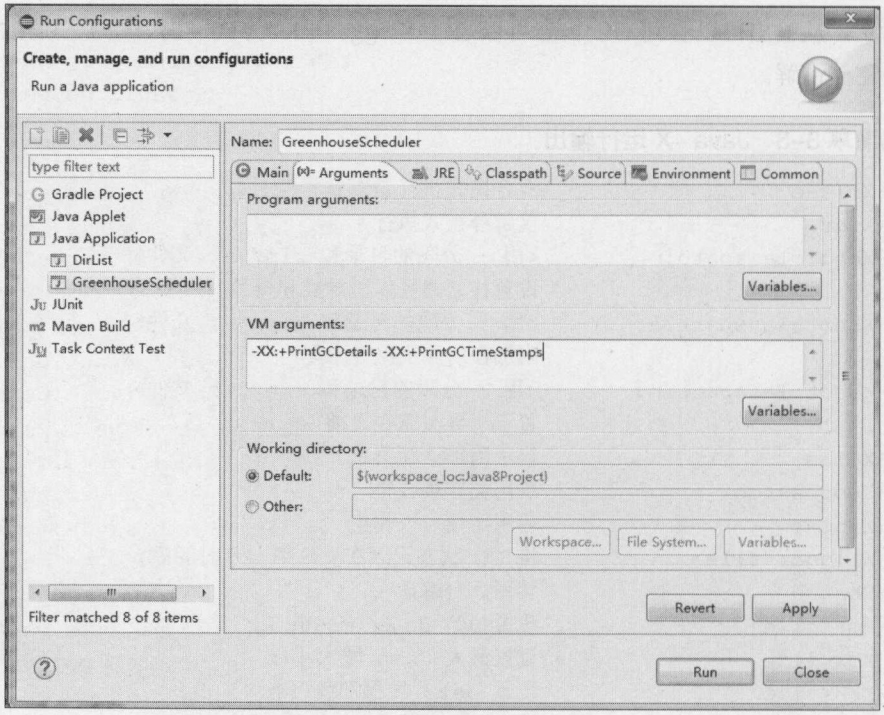


图 3-1　如何输出 GC 日志

然后单击 Run 按钮，可以看到在 Eclipse 的 Console[1] Tab 里面输出如代码清单 3-4 所示。

代码清单 3–4　–XX:+PrintGCDetails 运行输出

```
Heap
 PSYoungGen      total 17920K, used 5552K [0x00000000ec100000,
0x00000000ed500000, 0x0000000100000000)
  eden space 15360K, 36% used [0x00000000ec100000,0x00000000ec66c398,
0x00000000ed000000)
  from space 2560K, 0% used [0x00000000ed280000,0x00000000ed280000,
0x00000000ed500000)
  to   space 2560K, 0% used [0x00000000ed000000,0x00000000ed000000,
0x00000000ed280000)
 ParOldGen       total 40960K, used 0K [0x00000000c4200000, 0x00000000c6a00000,
0x00000000ec100000)
  object space 40960K, 0% used [0x00000000c4200000,0x00000000c4200000,
```

[1] 即控制台，所有的 IDE 工具都有控制台这个 Tab，Linux 操作系统的控制台就是 shell 脚本执行地点。

```
0x00000000c6a00000)
    Metaspace       used 3958K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

这里没有指定任何其他选项，所以输出全部都是按照 JDK 版本默认对应的 GC 选项。

我们对上面的输出进行逐条剖析。“Heap”容易理解，既然是 GC，对应的对手肯定包含 Java 堆[1]，所以第一句输出 Heap（堆）也在情理之中。前面讲过，年龄代由年轻代、老年代组成，年轻代又由 Eden 和 Survivor 区组成，Survivor 又可以进一步划分为 From 区和 To 区。在 JDK8 之前还有一个永久区（PermGen），JDK8 开始被元空间（Metaspace）取而代之，而无论永久区还是元空间，都是用于存放类信息的，所以必然会显示 Class Space。这些元素在 GC 日志里面都可以体现，这就是为什么我们在上面这么一段 GC 日志输出里面都看到了它们的身影：Heap、PSYoungGen、Eden、From、To、Metaspace，以及 Class Space，这些信息的详细介绍我们放在后面的选项中逐一深入进行。

2. –Xloggc

前面那个选项运行时我们是在 Eclipse 的 Console 页面看到 GC 日志输出的，如果想要以文件形式保存这些 GC 日志，可以在 Eclipse 的 VM 选项里输入 JVM 的运行选项 -XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log，运行后我们会发现在项目文件夹下面生成了一个 gc.log 文件，打开后看到输出的 GC 日志，如代码清单 3-5 所示。

代码清单 3–5　–Xloggc 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1504580k free), swap 7843228k(5006548k
free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
   Heap
    PSYoungGen      total 17920K, used 5552K [0x00000000ec100000, 0x00000000ed500000,
0x0000000100000000)
     eden space 15360K, 36% used [0x00000000ec100000,0x00000000ec66c070,
```

[1] 所有的对象都保存在 Java 堆，垃圾回收不可能避开这个区域。

```
0x00000000ed000000)
     from space 2560K, 0% used [0x00000000ed280000,0x00000000ed280000,
0x00000000ed500000)
     to   space 2560K, 0% used [0x00000000ed000000,0x00000000ed000000,
0x00000000ed280000)
    ParOldGen       total 40960K, used 0K [0x00000000c4200000, 0x00000000c6a00000,
0x00000000ec100000)
     object space 40960K, 0% used [0x00000000c4200000,0x00000000c4200000,
0x00000000c6a00000)
    Metaspace       used 3967K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

日志输出时首先输出操作系统、JRE 相关信息，代码清单 3-5 输出的是 64 位操作系统（windows-amd64），JRE[1]（1.8.0_101-b13）版本。

接下来是内存空间，物理内存是 3.7GB，其中 1.43GB 空闲，交换区[2]7GB，其中 4.7GB 空闲。

我们并没有指定过多的参数，JVM 帮助我们增加了一些默认参数，即 CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC。

-XX:InitialHeapSize 和-XX:MaxHeapSize 就是我们比较熟悉的-Xms 和-Xmx，它们允许我们指定 JVM 的初始和最大堆内存大小。

自动添加的-XX:+UseCompressedClassPointers、-XX:+UseCompressedOops 以及-XX:-UseLargePagesIndividualAllocation 这三个选项和 OOP 有关。OOP 的全称是 Ordinary Object Pointer，即普通对象指针。通常 64 位 JVM 消耗的内存会比 32 位的大 1.5 倍，这是因为对象指针在 64 位架构下，长度会翻倍（更宽的寻址）。对于那些将要从 32 位平台移植到 64 位的应用来说，平白无故多了 1/2 的内存占用，作为开发者一定不愿意看到这种场景。所以，从 JDK 1.6 update14 开始，64 bit JVM 正式支持了 -XX:+UseCompressedOops 这个可以压缩指针，起到节约内存占用的选项。*CompressedOops 的实现方式是在机器码中植入压缩与解压指令，可能会给* JVM 增加额外的开销。-XX:+UseCompressedClassPointers 选项是在 JDK8 出现的，也是在永久区消失之后出现的新的选项，主要用于对类的元数据进行压缩。-XX:-UseLargePagesIndividual-

[1] 即 Java Runtime Environment，Java 运行环境，运行 Java 程序所必需的环境的集合，包含 JVM 标准实现及 Java 核心类库。

[2] 即 SWAP，类似于虚拟内存，就是当内存不足时，把一部分硬盘空间虚拟成内存使用，从而解决内存容量不足的问题。

Allocation 和 Oops 是一起使用的，在大页内存使用发生时这个选项也会自动启用。

-XX:+UseParallelGC 表示当我们没有指定 GC 时，由于 JDK 采用的是 JDK8，所以默认采用的是 ParallelGC（第 2 章介绍过，可能从 JDK9 开始会默认为 G1 GC）。

3. –XX:+UseSerialGC

除了 G1 GC 之外，其他 GC 都不是这本书的重点，这里只是简单地回顾它们调用后的输出，也是为了和 G1 GC 进行对比，这样更加形象一些。

Serial GC 是 HotSpot 客户端模式 VM 的默认 GC，它是一种独占式的 GC，即运行过程中会导致应用程序暂停（停顿），并且由于它是单线程执行的，所以停顿时间会比较长。Serial GC 在老年代使用了一种叫作“Mark-Sweep-Compact”的算法，它适用于 CPU 核数较少且使用的内存空间较小的应用程序场景。如果你的应用程序场景使用少于 100MB 的内存空间，并且不在乎停顿时间较长的话，Oracle 公司的官网推荐使用这种单线程方式的垃圾收集器。即，较 G1 GC、Serial GC 使用了较少的内存[1]。

Serial GC 和 Parallel GC 的差别是一个采用单线程，另一个采用多线程的处理方式。Serial GC 和 Parallel GC 的差别如图 3-2 所示。

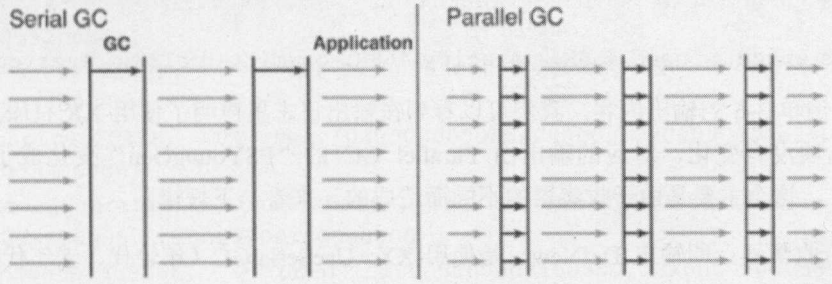


图 3-2　Serial GC 和 Parallel GC 的差别

继续前面类似的示例，这里使用 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+ UseSerialGC，运行后的输出如代码清单 3-6 所示。

代码清单 3–6　–XX:+UseSerialGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
```

[1] Charlie Hunt 认为 Serial GC 是所有 HotSpot 垃圾回收器里面最慢的，因为它使用内存跟踪。

```
(VS2010)
   Memory: 4k page, physical 3922532k(1875400k free), swap 7843228k(5597040k
free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseSerialGC
   Heap
    def new generation   total 18432K, used 5904K [0x00000000c4200000,
0x00000000c5600000, 0x00000000d8150000)
     eden space 16384K,  36% used [0x00000000c4200000, 0x00000000c47c43a0,
0x00000000c5200000)
     from space 2048K,   0% used [0x00000000c5200000, 0x00000000c5200000,
0x00000000c5400000)
     to   space 2048K,   0% used [0x00000000c5400000, 0x00000000c5400000,
0x00000000c5600000)
    tenured generation   total 40960K, used 0K [0x00000000d8150000,
0x00000000da950000, 0x0000000100000000)
     the space 40960K,   0% used [0x00000000d8150000, 0x00000000d8150000,
0x00000000d8150200, 0x00000000da950000)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved
1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

从代码清单 3-6 的输出内容，我们可以看到在输出日志里声明了使用-XX:+UseSerialGC 选项，其他选项没有变化，对应的输出由 Parallel GC 的“PSYoungGen”变化成了“def new generation”，这个主要是由于收集器的不同而造成的，来看一下规律。

- 串行收集器：即输出 DefNew，是使用-XX:+UseSerialGC（年轻代、老年代都使用串行回收收集器）运行后输出的。
- 并行收集器：即输出 ParNew，是使用-XX:+UseParNewGC（年轻代使用并行收集器，老年代使用串行回收收集器）或者-XX:+UseConcMarkSweepGC（年轻代使用并行收集器，老年代使用 CMS）运行后输出的。
- PSYoungGen：是使用-XX:+UseParallelOldGC（年轻代、老年代都使用并行回收收集器）或者-XX:+UseParallelGC（年轻代使用并行回收收集器，老年代使用串行回收收集器）运行输出的。
- Garbage-First heap：是使用-XX:+UseG1GC（G1 收集器）运行输出的。

一般来说，目前只在嵌入式应用产品场景下才使用 Serial GC。

4. -XX:+UseParNewGC

ParNew GC 是一种独占式的 GC，和 Serial GC 的区别是多线程 GC 执行。

还是继续实验，使用 VM 选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseParNewGC 运行后看看 GC 日志。运行输出如代码清单 3-7 所示。

代码清单 3-7　-XX:+UseParNewGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1857264k free), swap 7843228k(5596788k
free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParNewGC
   Heap
    par new generation   total 18432K, used 5906K [0x00000000c4200000,
0x00000000c5600000, 0x00000000d8150000)
     eden space 16384K,  36% used [0x00000000c4200000, 0x00000000c47c4800,
0x00000000c5200000)
     from space 2048K,   0% used [0x00000000c5200000, 0x00000000c5200000,
0x00000000c5400000)
     to   space 2048K,   0% used [0x00000000c5400000, 0x00000000c5400000,
0x00000000c5600000)
    tenured generation   total 40960K, used 0K [0x00000000d8150000,
0x00000000da950000, 0x0000000100000000)
      the space 40960K,   0% used [0x00000000d8150000, 0x00000000d8150000,
0x00000000d8150200, 0x00000000da950000)
    Metaspace       used 3950K, capacity 4716K, committed 4864K, reserved
1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

如前面所述，正确地输出了“par new generation”。除了正常的 GC 日志输出以外，在运行时还会看到 Eclipse 打印输出了警告信息，如代码清单 3-8 所示。

代码清单 3-8　-XX: + UseParNewGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM warning: Using the ParNew young collector
with the Serial old collector is deprecated and will likely be removed in a future
release
```

这段话说明了未来 ParNew 垃圾回收器不能再用了。

5. –XX:+UseParallelGC

这一段就是前面默认不选择 GC 时的输出，VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseParallelGC，运行输出如代码清单 3-9 所示。

代码清单 3-9 –XX:+UseParallelGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1875088k free), swap 7843228k(5614756k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
   Heap
    PSYoungGen      total 17920K, used 5550K [0x00000000ec100000, 0x00000000ed500000,
0x0000000100000000)
     eden space 15360K, 36% used [0x00000000ec100000,0x00000000ec66bb08,
0x00000000ed000000)
     from space 2560K, 0% used [0x00000000ed280000,0x00000000ed280000,
0x00000000ed500000)
     to   space 2560K, 0% used [0x00000000ed000000,0x00000000ed000000,
0x00000000ed280000)
    ParOldGen       total 40960K, used 0K [0x00000000c4200000, 0x00000000c6a00000,
0x00000000ec100000)
     object space 40960K, 0% used [0x00000000c4200000,0x00000000c4200000,
0x00000000c6a00000)
    Metaspace       used 3954K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

Parallel GC 是 JDK7 之后的默认 GC，它通过多线程方式减缓了独占式 GC 的副作用（停顿时间比较长）。年轻代和老年代在 Parallel GC 里都是并行执行且独占的，老年代也执行了压缩操作，这个压缩操作指的是移动存活对象到相邻位置，这样可以减少内存浪费，更好地实现内存布局。

Serial GC 使用单一线程执行 GC，而 Parallel GC 则使用多个线程并发执行，因此 Parallel GC 较 Serial GC 具有更快的回收速度。Parallel GC 适用于多核 CPU 且使用了较大内存空间的场景。

此外，Parallel GC 又被称为“高吞吐 GC（Throughput GC）”。

6. -XX:+UseParallelOldGC

Parallel Old GC 在 JDK 5 中被引入，与 Parallel GC 相比唯一的区别在于 Parallel Old GC 算法是为老年代设计的。它的执行过程分为三步，即标记（Mark）、总结（Summary）、压缩（Compaction）。其中 Summary 步骤为存活的对象在已执行过 GC 的空间上标出位置，因此与 Mark-Sweep-Compact 算法中的 Sweep 步骤有所区别，并需要一些复杂步骤才能完成。

在 JDK7U45 之前，我们在使用 Parallel GC 时是区分年轻代和老年代的，即老年代并行回收收集器需要通过设置 UseParallelOldGC 来启动，在 JDK7U45 之后两者合并了。

继续之前的实验，设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseParallelOldGC，运行输出如代码清单 3-10 所示。

代码清单 3-10　-XX:+UseParallelOldGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1461196k free), swap 7843228k(4975780k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:-UseLargePagesIndividualAllocation -XX:+UseParallelOldGC
   Heap
    PSYoungGen      total 17920K, used 5550K [0x00000000ec100000,
0x00000000ed500000, 0x0000000100000000)
     eden space 15360K, 36% used [0x00000000ec100000,0x00000000ec66ba68,
0x00000000ed000000)
     from space 2560K, 0% used [0x00000000ed280000,0x00000000ed280000,
0x00000000ed500000)
     to   space 2560K, 0% used [0x00000000ed000000,0x00000000ed000000,
0x00000000ed280000)
    ParOldGen       total 40960K, used 0K [0x00000000c4200000, 0x00000000c6a00000,
0x00000000ec100000)
     object space 40960K, 0% used [0x00000000c4200000,0x00000000c4200000,
0x00000000c6a00000)
    Metaspace       used 3955K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

7. –XX:+UseConcMarkSweepGC

从图 3-3 可以看出并发标记-清理（Concurrent Mark-Sweep） GC 相比其他 GC 都要复杂。初始标记（Initial Mark）比较简单，只有靠近类加载器的存活对象会被标记，因此停顿时间（Stop-the-world pause）比较短暂。在并发标记（Concurrent Mark）阶段，刚被确认和标记过的存活对象所关联的对象将会被跟踪和检测存活状态。此步骤的不同之处在于有多个线程并行处理此过程。在重标记（Remark）阶段，由并发标记所关联的新增或中止的对象会被检测。在最后的并发清理（Concurrent Sweep）阶段，垃圾回收过程被真正地执行。在垃圾回收执行过程中，其他线程依然可以保持并行执行。受益于 CMS GC 的执行方式，在 GC 期间系统中断的时间非常短暂（G1 GC 也采纳了这种设计方案）。此外，CMS GC 也被称为低延迟 GC，适用于所有应用对响应时间要求比较严格的场景。

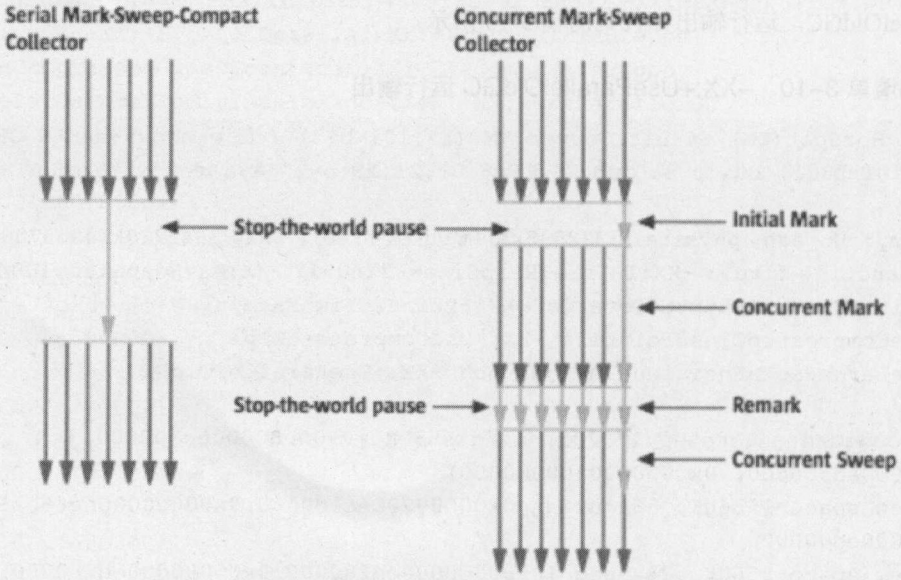


图 3-3　CMS VS Serial GC 执行顺序图

继续实验，设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+ UseConcMarkSweepGC。运行输出如代码清单 3-11 所示。

代码清单 3–11　–XX:+UseConcMarkSweepGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
```

```
   Memory: 4k page, physical 3922532k(1873928k free), swap 7843228k(5613224k
free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:MaxNewSize=174485504 -XX:MaxTenuringThreshold=6 -XX:OldPLABSize=16
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:+UseConcMarkSweepGC -XX:-UseLargePagesIndividualAllocation
-XX:+UseParNewGC
   Heap
    par new generation   total 18432K, used 5904K [0x00000000c4200000,
0x00000000c5600000, 0x00000000ce860000)
     eden space 16384K,  36% used [0x00000000c4200000, 0x00000000c47c4120,
0x00000000c5200000)
     from space 2048K,   0% used [0x00000000c5200000, 0x00000000c5200000,
0x00000000c5400000)
     to   space 2048K,   0% used [0x00000000c5400000, 0x00000000c5400000,
0x00000000c5600000)
    concurrent mark-sweep generation total 40960K, used 0K [0x00000000ce860000,
0x00000000d1060000, 0x0000000100000000)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

在使用 CMS GC 之前需要对系统做全面的分析。另外，为了避免过多的内存碎片而需要执行压缩任务时，CMS GC 会比任何其他 GC 带来更多的 Stop-the-World 时间，所以需要分析和判断压缩任务执行的频率及其耗时情况。

8. –XX:+UseG1GC

前面七条除了前两条是通用选项以外，其他都是针对指定 GC 的方法介绍，现在终于进入到这本书的重点 G1 GC 了。

使用 UseG1GC 这个选项要求 JDK7 或者 JDK8 对应的 JVM 采用 G1 GC，据说从 JDK9 开始默认 GC 会变更为 G1 GC（现在是 Parallel GC），但也不确定。

我们使用 VM 选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC，日志输出如代码清单 3-12 所示。

代码清单 3-12　–XX:+UseG1GC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
```

```
   Memory: 4k page, physical 3922532k(1585004k free), swap 7843228k(5137568k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 3 young (3072K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

G1 GC 的日志输出和其他 GC 有所不同，它更加简洁。这里没有进入到一个评估阶段，评估阶段就是确认有多少对象需要被回收，通常是针对年轻代或者年轻代+老年代的。从上面的输出我们可以看出一共有 60MB（61440/1024）的堆内存空间，其中使用了 2MB。Region 是每个 1MB，有 3 个年轻代 Region。元数据空间的使用情况也做了相应的介绍，使用了 3.8MB，可用的为 4.6MB。

9. –XX:+PrintGCApplicationStoppedTime

如果使用该选项，会输出 GC 造成应用程序暂停的时间。一般和-XX:+ PrintGCApplication-ConcurrentTime 组合起来一起使用，这样比较有利于查看输出。

使用 VM 选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+ Print-GCApplicationStoppedTime -XX:+PrintGCApplicationConcurrentTime 运行应用程序，日志输出如代码清单 3-13 所示。

代码清单 3–13　–XX:+ PrintGCApplicationStoppedTime 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0 (VS2010)
   Memory: 4k page, physical 3922532k(1449048k free), swap 7843228k(4974672k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCApplicationConcurrentTime
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.159: Application time: 0.8766273 seconds
   1.160: Total time for which application threads were stopped: 0.0001140
seconds, Stopping threads took: 0.0000351 seconds
   4.159: Application time: 2.9998299 seconds
```

```
   4.160: Total time for which application threads were stopped: 0.0002126
seconds, Stopping threads took: 0.0000724 seconds
   4.418: Application time: 0.2587571 seconds
   4.419: Total time for which application threads were stopped: 0.0001491
seconds, Stopping threads took: 0.0000381 seconds
   5.462: Application time: 1.0433280 seconds
   5.462: Total time for which application threads were stopped: 0.0000664
seconds, Stopping threads took: 0.0000360 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 3 young (3072K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
   5.463: Application time: 0.0007787 seconds
```

看一下这两行输出：

```
   1.159: Application time: 0.8766273 seconds
   1.160: Total time for which application threads were stopped: 0.0001140
seconds, Stopping threads took: 0.0000351 seconds
```

这里表示应用程序执行了0.87s，GC线程造成的停顿时间大约为0.0001140s。由于程序在运行过程中进行了多次回收，所以你看到这里有多次的时间打印。如果发现某个时间很长，那你就要关注代码和设计了，分析哪里可能出现了实现或者设计弱点（不一定是缺陷），再根据实际情况进行优化，也许是代码逻辑复杂造成的，也可能是代码编写时频繁创建对象造成的。

这里应用程序会被暂停是由于G1 GC针对年轻代（有时候是年轻代+老年代）有一个评估阶段，这个评估阶段实质上是在做数据拷贝，既然是拷贝，就一定需要一个基准点[1]，那么为了维护这个基准点，需要设置对应的应用程序暂停时间，这个时间段就称为保护点（safepoint），这和Oracle的checkPoint很像。

10. -XX:ConcGCThreads

这个选项用来设置与Java应用程序线程并行执行的GC线程数量，默认为GC独占时运行线程的1/4。这个选项设置过大会导致Java应用程序可以使用的CPU资源减少，如果小一点则会对应用程序有利，但是过小就会增加GC并行循环的执行时间，反过来减少Java应用程序的

[1] 即大家都认为这个时间点的数据是完好无缺的，永远不要认为软件应该无缝隙保证数据的完整性，这个和CAP原理相违背，一定是通过其他方式来确保数据的原子性、完整性。

运行时间（因为独占期时间拉长）。

设置一个比较大的 ConcGCThread 值，如-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:ConcGCThreads=4，会看到 JVM 抛出如下用红色标记的错误，表明在初始化 JVM 时出错，不能够创建并行标记阶段，并且最终拒绝执行程序。这个错误表明当前机器只允许启动两个并行 GC 线程，这是根据运行程序所在机器的 CPU 的核数计算出来的。

```
   Error occurred during initialization of VM
   Could not create/initialize ConcurrentMark
   Java HotSpot(TM) 64-Bit Server VM warning: Can't have more ConcGCThreads (4)
than ParallelGCThreads (2).
```

把 ConcGCThreads 的值改为 2 之后，即-XX:ConcGCThreads=2，运行输出如代码清单 3-14 所示。

代码清单 3-14　-XX:ConcGCThreads 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE (1.8.0_
101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0 (VS2010)
   Memory: 4k page, physical 3922532k(1632868k free), swap 7843228k(5179308k
free)
   CommandLine flags: -XX:ConcGCThreads=2 -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   1.110: Total time for which application threads were stopped: 0.0002820
seconds, Stopping threads took: 0.0000720 seconds
   4.115: Total time for which application threads were stopped: 0.0002280
seconds, Stopping threads took: 0.0000750 seconds
   4.346: Total time for which application threads were stopped: 0.0003227
seconds, Stopping threads took: 0.0000784 seconds
   5.396: Total time for which application threads were stopped: 0.0000806
seconds, Stopping threads took: 0.0000416 seconds
   Heap
    garbage-first heap   total 61440K, used 3072K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

如果不设置并行标记线程的数量，默认情况下采用下面这个公式计算出来：

```
   ConcGCThreads=Max((ParallelGCThreads#+2)/4,1)
```

另外，如果设置的值不在有效范围内，那么就会抛出如下错误：

```
   Invalid number of concurrent marking threads:  -XX:ConcGCThreads=
[specified_value]
```

11. –XX:G1HeapRegionSize

这是 G1 GC 独有的选项，它是专门针对 Region 这个概念的对应设置选项，后续 GC 应该会继续采用 Region 这个概念。Region 的大小默认为堆大小的 1/2000，也可以设置为 1MB、2MB、4MB、8MB、16MB，以及 32MB，这六个划分档次。

增大 Region 块的大小有利于处理大对象。前面介绍过，大对象没有按照普通对象方式进行管理和分配空间，如果增大 Region 块的大小，则一些原本走特殊处理通道的大对象就可以被纳入普通处理通道了。这就好比我们在机场安检，飞行员、空姐可以走特殊通道，乘客如果也搞特殊化，一部分人去特殊通道处理，那么特殊通道就得增加几个，相应的普通通道就得减少了，对效率就起了降低作用。反之，如果 Region 大小设置过小，则会降低 G1 的灵活性，对于各个年龄代的大小都会造成分配问题。

继续之前的类似例子，配置运行选项，这里设置每个 Region 的大小为 32MB，-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=32M

GC 日志输出如代码清单 3-15 所示。

代码清单 3–15　–XX:G1HeapRegionSize 运行输出 1

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1608364k free), swap 7843228k(5151084k
free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=33554432
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.191: Total time for which application threads were stopped: 0.0001496
seconds, Stopping threads took: 0.0000411 seconds
   3.191: Total time for which application threads were stopped: 0.0000694
seconds, Stopping threads took: 0.0000317 seconds
   4.403: Total time for which application threads were stopped: 0.0001723
```

```
seconds, Stopping threads took: 0.0000386 seconds
   5.402: Total time for which application threads were stopped: 0.0000754
seconds, Stopping threads took: 0.0000441 seconds
   5.413: Total time for which application threads were stopped: 0.0000639
seconds, Stopping threads took: 0.0000334 seconds
   Heap
    garbage-first heap   total 65536K, used 0K [0x00000000c4000000,
0x00000000c6000010, 0x0000000100000000)
     region size 32768K, 1 young (32768K), 0 survivors (0K)
    Metaspace       used 3955K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

从上面的输出可以看到，一个 Region 的大小是 32MB（32768/1024）。换成 1MB 之后，日志输出如代码清单 3-16 所示，最大的区别是 Region 的大小变成了 1MB（1024/1024），有 4 个年轻代区间。

代码清单 3-16　-XX:G1HeapRegionSize 运行输出 2

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1614848k free), swap 7843228k(5159776k
free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=1048576
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.080: Total time for which application threads were stopped: 0.0001303
seconds, Stopping threads took: 0.0000356 seconds
   4.080: Total time for which application threads were stopped: 0.0001346
seconds, Stopping threads took: 0.0000523 seconds
   4.374: Total time for which application threads were stopped: 0.0001740
seconds, Stopping threads took: 0.0000437 seconds
   5.348: Total time for which application threads were stopped: 0.0000737
seconds, Stopping threads took: 0.0000360 seconds
   Heap
    garbage-first heap   total 61440K, used 3072K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3964K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

换成 2MB 之后，调整也是对应的，如代码清单 3-17 所示。

代码清单 3-17　-XX:G1HeapRegionSize 运行输出 3

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1622124k free), swap 7843228k(5165420k
free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.114: Total time for which application threads were stopped: 0.0001307
seconds, Stopping threads took: 0.0000347 seconds
   4.118: Total time for which application threads were stopped: 0.0001204
seconds, Stopping threads took: 0.0000390 seconds
   4.423: Total time for which application threads were stopped: 0.0001419
seconds, Stopping threads took: 0.0000334 seconds
   5.450: Total time for which application threads were stopped: 0.0000759
seconds, Stopping threads took: 0.0000369 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3975K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

这里设置了几个不同的值作为测试，实际情况下最好根据自己的实际内存大小进行设置。现在的服务器动不动就 128GB 以上的内存，这个选项用好了会很受益，但用差了性能下降也会很严重，因为需要打印 GC 的中断时间，所以需要综合来看问题。

12. -XX:G1HeapWastePercent

这个选项控制 G1 GC 不会回收的空闲内存比例，默认是堆内存的 5%。G1 GC 在回收过程中会回收所有 Region 的内存，并持续地做这个工作直到空闲内存比例达到设置的这个值为止，所以对于设置了较大值的堆内存来说，需要采用比较低的比例，这样可以确保较小部分的内存不被回收。这个很容易理解，城市越大就越容易出现一些死角，出于性能的原因可以不去关注那里，但是这个比例不能大。还是应该处处做到 100 分。

设置不回收的比例为 99%，设备选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=99，日志输出如代码清单 3-18 所示。

代码清单 3-18　-XX:G1HeapWastePercent 运行输出 1

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1961388k free), swap 7843228k(5602892k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=99 -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   1.118: Total time for which application threads were stopped: 0.0001457
seconds, Stopping threads took: 0.0000373 seconds
   4.149: Total time for which application threads were stopped: 0.0003476
seconds, Stopping threads took: 0.0002169 seconds
   4.305: Total time for which application threads were stopped: 0.0001590
seconds, Stopping threads took: 0.0000403 seconds
   5.382: Total time for which application threads were stopped: 0.0000896
seconds, Stopping threads took: 0.0000381 seconds
   Heap
    garbage-first heap   total 61440K, used 0K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 1 young (2048K), 0 survivors (0K)
    Metaspace       used 3947K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

另一个极端，设置为 1%，运行输出如代码清单 3-19 所示。

代码清单 3-19　-XX:G1HeapWastePercent 运行输出 2

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1965444k free), swap 7843228k(5606132k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=1 -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
```

```
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   1.095: Total time for which application threads were stopped: 0.0002850
seconds, Stopping threads took: 0.0000729 seconds
   4.139: Total time for which application threads were stopped: 0.0001607
seconds, Stopping threads took: 0.0000519 seconds
   4.250: Total time for which application threads were stopped: 0.0019170
seconds, Stopping threads took: 0.0018026 seconds
   5.347: Total time for which application threads were stopped: 0.0001196
seconds, Stopping threads took: 0.0000669 seconds
   Heap
    garbage-first heap   total 61440K, used 0K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 1 young (2048K), 0 survivors (0K)
    Metaspace       used 3950K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

这两个设置都是比较极端的，一般情况下我们不会做出调整，即采用 5%的默认值。当设置为 1%时，本例的差别不是很明显，但是 GC 线程造成的应用程序暂停时间已经比 99%时有了明显的增长。

13. –XX:G1MixedGCCountTarget

老年代 Region 的回收时间通常来说比年轻代 Region 稍长一些，这个选项可以设置一个并行循环之后启动多少个混合 GC，默认值是 8 个。设置一个比较大的值可以让 G1 GC 在老年代 Region 回收时多花一些时间，如果一个混合 GC 的停顿时间很长，说明它要做的事情很多，所以可以增大这个值的设置，但是如果这个值过大，也会造成并行循环等待混合 GC 完成的时间相应的增加。

还是继续实验，设置选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+ UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize= 2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=80，日志输出如代码清单 3-20 所示。

代码清单 3–20　–XX:G1MixedGCCountTarget 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0 (VS2010)
   Memory: 4k page, physical 3922532k(1963596k free), swap 7843228k(5602320k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=80
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
```

```
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.068: Total time for which application threads were stopped: 0.0001230
seconds, Stopping threads took: 0.0000321 seconds
   4.098: Total time for which application threads were stopped: 0.0001286
seconds, Stopping threads took: 0.0000446 seconds
   4.207: Total time for which application threads were stopped: 0.0001809
seconds, Stopping threads took: 0.0000369 seconds
   5.317: Total time for which application threads were stopped: 0.0000716
seconds, Stopping threads took: 0.0000386 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

这个选项和 G1MixedGCLIveThresholdPercent 选项（默认值为 65%，指的是混合回收阶段回收老年代区间的上限阈值）有直接关联，G1MixedGCLIveThresholdPercent 设置完毕后，对存活数据回收的上限为 G1MixedGCLIveThresholdPercent 的旧区域执行混合垃圾回收的目标次数，这个选项会自动调整为满足配置的阈值，以便能够快速回收对象。

14. -XX:+G1PrintRegionLivenessInfo

由于开启这个选项会在标记循环阶段完成之后输出详细信息，专业一点的叫法是诊断选项[1]，所以在使用前需要开启选项 UnlockDiagnosticVMOptions。这个选项启用后会打印堆内存内部每个 Region 里面的存活对象信息，这些信息包括使用率、RSet 大小、回收一个 Region 的价值（Region 内部回收价值评估，即性价比）。

这个选项输出的信息对于调试堆内 Region 是很有效的，不过对于一个很大的堆内存来说，由于每个 Region 信息都输出了，所以信息量也是挺大的。还是继续实验，设置选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget= 10 -XX:+G1PrintRegionLivenessInfo。

直接运行会发现抛出错误：“Error: VM option 'G1PrintRegionLivenessInfo' is diagnostic and

[1] 即 Diagnostic VM Option。

must be enabled via -XX:+UnlockDiagnosticVMOptions.”。

注意，-XX:+UnlockDiagnosticVMOptions 必须放在-XX:+G1PrintRegionLivenessInfo 的前面，证明 VM 的选项执行顺序也是串行的，这就是为什么抛出了上面的错误，调整一下：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo。

运行程序，日志输出如代码清单 3-21 所示。

代码清单 3–21　–XX:G1PrintRegionLivenessInfo 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1661092k free), swap 7843228k(5133168k
free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10
-XX:+G1PrintRegionLivenessInfo -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UnlockDiagnosticVMOptions
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.165: Total time for which application threads were stopped: 0.0001389
seconds, Stopping threads took: 0.0000360 seconds
   4.165: Total time for which application threads were stopped: 0.0001243
seconds, Stopping threads took: 0.0000416 seconds
   4.483: Total time for which application threads were stopped: 0.0001551
seconds, Stopping threads took: 0.0000377 seconds
   5.460: Total time for which application threads were stopped: 0.0006171
seconds, Stopping threads took: 0.0005734 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3950K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

15. –XX:G1ReservePercent

每个年龄代都会有一些对象可以进入下一个阶段，为了确保这个提升过程正常完成，我们允许 G1 GC 保留一些内存，这样就可以避免出现“to space exhausted”错误，这个选项就是为了这个用途。

这个选项默认保留堆内存的 10%。注意，这个预留内存空间不能用于年轻代。

对于一个拥有大内存的堆内存来说，这个值不能过大，因为它不能用于年轻代，这就意味着年轻代可用内存降低了。减小这个值有助于给年轻代留出更大的内存空间、更长的 GC 时间，这对提升性能吞吐量有好处。

继续实验，先测试一个极端值，这里配置 G1ReservePercent 为 100%，配置选项-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=100。我们可以看到运行时抛出了异常：“Java HotSpot(TM) 64-Bit Server VM warning: G1ReservePercent is set to a value that is too large, it's been updated to 50.”。

这里提示我们，最大的保留空间不能超过 50%，指的是堆内存的 50%。减少为 50%，运行后日志输出如代码清单 3-22 所示。

代码清单 3–22 –XX: G1ReservePercent 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1661008k free), swap 7843228k(5140300k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10
-XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=50
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UnlockDiagnosticVMOptions -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   1.180: Total time for which application threads were stopped: 0.0001783
seconds, Stopping threads took: 0.0000531 seconds
   4.180: Total time for which application threads were stopped: 0.0001046
seconds, Stopping threads took: 0.0000330 seconds
   4.406: Total time for which application threads were stopped: 0.0001414
```

```
seconds, Stopping threads took: 0.0000356 seconds
   5.478: Total time for which application threads were stopped: 0.0000767
seconds, Stopping threads took: 0.0000343 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3967K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

16. –XX:+G1SummarizeRSetStats

和 G1PrintRegionLivenessInfo 选项一样，这个选项也是一个诊断选项，所以也需要开启 UnlockDiagnosticVMOptions 选项后才能使用，这也就意味着-XX:+UnlockDiagnosticVMOptions 选项需要放在-XX:+G1SummarizeRSetStats 选项的前面。

这个选项和-XX:G1SummarizeRSetStatsPeriod 一起使用的时候会阶段性地打印 RSets 的详细信息，这有助于找到 RSet 里面存在的问题。采用选项配置：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10 -XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1SummarizeRSetStats。运行后日志输出如代码清单 3-23 所示。

代码清单 3–23　–XX: +G1SummarizeRSetStats 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1692124k free), swap 7843228k(5178744k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10
-XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10
-XX:+G1SummarizeRSetStats -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UnlockDiagnosticVMOptions
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.086: Total time for which application threads were stopped: 0.0001337
seconds, Stopping threads took: 0.0000360 seconds
```

```
   4.086: Total time for which application threads were stopped: 0.0001037
seconds, Stopping threads took: 0.0000326 seconds
   4.342: Total time for which application threads were stopped: 0.0001534
seconds, Stopping threads took: 0.0000373 seconds
   5.372: Total time for which application threads were stopped: 0.0000746
seconds, Stopping threads took: 0.0000296 seconds
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3965K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
    Cumulative RS summary

    Recent concurrent refinement statistics
     Processed 0 cards
     Of 0 completed buffers:
            0 (  0.0%) by concurrent RS threads.
            0 (  0.0%) by mutator threads.
     Did 0 coarsenings.
     Concurrent RS threads times (s)
           0.00     0.00
     Concurrent sampling threads times (s)
           0.00

    Current rem set statistics
     Total per region rem sets sizes = 154K. Max = 5K.
           10K (  7.0%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
          143K ( 93.0%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      Static structures = 7K, free_lists = 0K.
       0 occupied cards represented.
            0 (  0.0%) entries by 2 Young regions
            0 (  0.0%) entries by 0 Humonguous regions
            0 (  0.0%) entries by 28 Free regions
            0 (  0.0%) entries by 0 Old regions
       Region with largest rem set = 29:(E)[0x00000000c7c00000,0x00000000c7e00000,
0x00000000c7e00000], size = 5K, occupied = 0K.
     Total heap region code root sets sizes = 1K.  Max = 0K.
           0K ( 57.9%) by 2 Young regions
```

```
            0K (  0.0%) by 0 Humonguous regions
            0K ( 42.1%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
       22 code roots represented.
            22 (100.0%) elements by 2 Young regions
             0 (  0.0%) elements by 0 Humonguous regions
             0 (  0.0%) elements by 28 Free regions
             0 (  0.0%) elements by 0 Old regions
   Region with largest amount of code roots = 29:(E)[0x00000000c7c00000,
0x00000000c7e00000,0x00000000c7e00000], size = 0K, num_elems = 0.
```

从上面的输出可以看到，打印出了每个 Region 对应的 RSet 的详细信息，这些信息的解释放在第 5 章讲解。

17. –XX:+G1TraceConcRefinement

这是一个诊断选项。如果启动这个诊断选项，那么并行 Refinement 线程相关的信息会被打印。注意，线程启动和结束时的信息都会被打印。

这里提到了 Refinement 线程，我们来提前梳理这个概念。请看每一代 GC 对应的 GC 线程，如表 3-1 所示。

表 3-1　垃圾收集器对应的GC线程

Garbage Collector	Worker Threads Used
Parallel GC	ParallelGCThreads
CMS GC	ParallelGCThreads ConcGCThreads
G1 GC	ParallelGCThreads ConcGCThreads G1ConcRefinementThreads

上面列出了三类 GC 线程，分别是 ParallelGCThreads、ConcGCThreads 和 G1ConcRefinementThreads。在第 4 章会介绍到 G1 GC 从初始标记阶段开始，一直到并行清除阶段结束，其中一些阶段 GC 是可以和应用程序共同运行的，另外一些阶段是 GC 独占的，那么这三个线程就有所区别了。关于这三个线程的区别，请看表 3-2。

表 3-2　GC线程定义图

名称	选项控制	作用
ParallelGC Thread	-XX:ParallelGCThreads	GC的并行工作线程，专门用于独占阶段的工作，比如拷贝存活对象

续表

名称	选项控制	作用
ParallelMarkingThreads	-XX:ConcGCThreads	并行标记阶段的并行线程，它由一个主控（Master）线程和一些工作（Worker）线程组成，可以和应用程序并行执行
G1ConcurrentRefinementThreads	-XX:G1ConcRefinementThreads	和应用程序一起运行，用于更新RSet。如果Concurrent-RefinementThreads 没有设置，那么默认为ParallelGCThreads + 1

继续实验，设置VM选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+ UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:ConcGCThreads=1 -XX:G1HeapRegionSize= 2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10 -XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1SummarizeRSetStats -XX:+G1TraceConcRefinement。运行输出如代码清单 3-24 所示。

代码清单 3-24 -XX: + G1TraceConcRefinement 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1895664k free), swap 7843228k(5633652k free)
   CommandLine flags: -XX:ConcGCThreads=1 -XX:G1HeapRegionSize=2097152
-XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10
-XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10
-XX:+G1SummarizeRSetStats -XX:G1SummarizeRSetStatsPeriod=10
-XX:+G1TraceConcRefinement -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UnlockDiagnosticVMOptions
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.095: Total time for which application threads were stopped: 0.0001307
seconds, Stopping threads took: 0.0000351 seconds
   4.095: Total time for which application threads were stopped: 0.0001110
seconds, Stopping threads took: 0.0000377 seconds
   4.370: Total time for which application threads were stopped: 0.0003351
seconds, Stopping threads took: 0.0000793 seconds
   5.347: Total time for which application threads were stopped: 0.0000716
seconds, Stopping threads took: 0.0000394 seconds
   G1-Refine-stop
   G1-Refine-stop
   G1-Refine-stop
```

```
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3949K, capacity 4716K, committed 4864K, reserved
1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
    Cumulative RS summary

    Recent concurrent refinement statistics
     Processed 0 cards
     Of 0 completed buffers:
            0 (  0.0%) by concurrent RS threads.
            0 (  0.0%) by mutator threads.
     Did 0 coarsenings.
     Concurrent RS threads times (s)
           0.00     0.00
     Concurrent sampling threads times (s)
           0.00

    Current rem set statistics
     Total per region rem sets sizes = 154K. Max = 5K.
           10K (  7.0%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
          143K ( 93.0%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      Static structures = 7K, free_lists = 0K.
       0 occupied cards represented.
            0 (  0.0%) entries by 2 Young regions
            0 (  0.0%) entries by 0 Humonguous regions
            0 (  0.0%) entries by 28 Free regions
            0 (  0.0%) entries by 0 Old regions
       Region with largest rem set = 29:(E)[0x00000000c7c00000,0x00000000c7e00000,
0x00000000c7e00000], size = 5K, occupied = 0K.
     Total heap region code root sets sizes = 1K.  Max = 0K.
            0K ( 56.9%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
            0K ( 43.1%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
       21 code roots represented.
            21 (100.0%) elements by 2 Young regions
```

```
            0 (  0.0%) elements by 0 Humonguous regions
            0 (  0.0%) elements by 28 Free regions
            0 (  0.0%) elements by 0 Old regions
       Region with largest amount of code roots = 29:(E)[0x00000000c7c00000,
0x00000000c7e00000,0x00000000c7e00000], size = 0K, num_elems = 0.
```

我们可以看到，G1-Refine-stop 这一串单词出现了，具体还是在第 4 章讲解。

18. –XX:+G1UseAdaptiveConcRefinement

这个选项默认是开启的。它会动态地对每一次 GC 中-XX:G1ConcRefinementGreenZone、-XX:G1ConcRefinementYellowZone、-XX:G1ConcRefinementRedZone 的值进行重新计算。

并行 Refinement 线程是持续运行的，并且会随着 update log buffer 积累的数量而动态调节。前面说到的三个配置选项-XX:G1ConcRefinementGreenZone、-XX:G1ConcRefinementYellowZone、-XX:G1ConcRefinementRedZone，是被用来根据不同的 buffer 使用不同的 Refinement 线程，目的就是为了保证 Refinement 线程一定要尽可能地跟上 update log buffer 产生的步伐。但是这个 Refinement 线程不是无限增加的，一旦出现 Refinement 线程跟不上 update log buffer 产生的速度、update log buffer 开始出现积压的情况，Mutator 线程（即应用业务线程）就会协助 Refinement 线程执行 RSet 的更新工作。这个 Mutator 线程实际上就是应用业务线程，当业务线程去参与 RSet 修改时，系统性能一定会受到影响，所以需要尽力去避免这种状况。

回到 G1UseAdaptiveConcRefinement 这个选项，还是继续实验，设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10 -XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1SummarizeRSetStats -XX:+G1TraceConcRefinement -XX:+G1UseAdaptiveConcRefinement。运行输出如代码清单 3-25 所示。

代码清单 3–25　–XX:+G1UseAdaptiveConcRefinement 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1888412k free), swap 7843228k(5628340k
free)
   CommandLine flags: -XX:G1HeapRegionSize=2097152 -XX:G1HeapWastePercent=5
-XX:G1MixedGCCountTarget=10 -XX:+G1PrintRegionLivenessInfo
-XX:G1ReservePercent=10 -XX:+G1SummarizeRSetStats
```

```
-XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1TraceConcRefinement
-XX:+G1UseAdaptiveConcRefinement -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UnlockDiagnosticVMOptions
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.058: Total time for which application threads were stopped: 0.0002893
seconds, Stopping threads took: 0.0000746 seconds
   4.058: Total time for which application threads were stopped: 0.0001037
seconds, Stopping threads took: 0.0000339 seconds
   4.303: Total time for which application threads were stopped: 0.0001680
seconds, Stopping threads took: 0.0000420 seconds
   5.303: Total time for which application threads were stopped: 0.0002306
seconds, Stopping threads took: 0.0001971 seconds
   5.312: Total time for which application threads were stopped: 0.0000673
seconds, Stopping threads took: 0.0000373 seconds
   G1-Refine-stop
   G1-Refine-stop
   G1-Refine-stop
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3951K, capacity 4716K, committed 4864K, reserved
1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
    Cumulative RS summary

    Recent concurrent refinement statistics
     Processed 0 cards
     Of 0 completed buffers:
            0 (  0.0%) by concurrent RS threads.
            0 (  0.0%) by mutator threads.
     Did 0 coarsenings.
     Concurrent RS threads times (s)
           0.00    0.00
     Concurrent sampling threads times (s)
           0.00

    Current rem set statistics
     Total per region rem sets sizes = 154K. Max = 5K.
```

```
           10K (  7.0%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
          143K ( 93.0%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      Static structures = 7K, free_lists = 0K.
       0 occupied cards represented.
             0 (  0.0%) entries by 2 Young regions
             0 (  0.0%) entries by 0 Humonguous regions
             0 (  0.0%) entries by 28 Free regions
             0 (  0.0%) entries by 0 Old regions
      Region with largest rem set = 29:(E)[0x00000000c7c00000,0x00000000c7e00000,
0x00000000c7e00000], size = 5K, occupied = 0K.
     Total heap region code root sets sizes = 0K.  Max = 0K.
            0K ( 55.9%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
            0K ( 44.1%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      20 code roots represented.
            20 (100.0%) elements by 2 Young regions
             0 (  0.0%) elements by 0 Humonguous regions
             0 (  0.0%) elements by 28 Free regions
             0 (  0.0%) elements by 0 Old regions
      Region with largest amount of code roots = 29:(E)[0x00000000c7c00000,
0x00000000c7e00000,0x00000000c7e00000], size = 0K, num_elems = 0.
```

19. –XX:GCTimeRatio

这个选项代表 Java 应用线程花费的时间与 GC 线程花费时间的比率。通过这个比率值可以调节 Java 应用线程或者 GC 线程的工作时间，保障两者的执行时间。

HotSpot VM 转换这个值为一个百分比，公式是 100/(1+GCTimeRatio)，默认值是 9，表示花费在 GC 工作量上的时间占总时间的 10%。Parallel GC 里这个值为 99，即代表 GC 只有 1%的时间。设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -XX:+UseG1GC -XX:+ PrintGCApplication-StoppedTime -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent=5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10 -XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1SummarizeRSetStats -XX:+G1TraceConcRefinement -XX:+G1UseAdaptiveConcRefinement -XX:GCTimeRatio=5。运行输出结果如代码清单 3-26 所示。

代码清单 3-26　-XX:GCTimeRatio 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1884916k free), swap 7843228k(5614188k
free)
   CommandLine flags: -XX:G1HeapRegionSize=2097152 -XX:G1HeapWastePercent=5
-XX:G1MixedGCCountTarget=10 -XX:+G1PrintRegionLivenessInfo
-XX:G1ReservePercent=10 -XX:+G1SummarizeRSetStats
-XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1TraceConcRefinement
-XX:+G1UseAdaptiveConcRefinement -XX:GCTimeRatio=90
-XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UnlockDiagnosticVMOptions -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   1.160: Total time for which application threads were stopped: 0.0003446
seconds, Stopping threads took: 0.0000986 seconds
   4.160: Total time for which application threads were stopped: 0.0002409
seconds, Stopping threads took: 0.0000896 seconds
   4.376: Total time for which application threads were stopped: 0.0012300
seconds, Stopping threads took: 0.0000827 seconds
   5.406: Total time for which application threads were stopped: 0.0001153
seconds, Stopping threads took: 0.0000849 seconds
   G1-Refine-stop
   G1-Refine-stop
   G1-Refine-stop
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
    Cumulative RS summary

    Recent concurrent refinement statistics
     Processed 0 cards
     Of 0 completed buffers:
            0 (  0.0%) by concurrent RS threads.
            0 (  0.0%) by mutator threads.
     Did 0 coarsenings.
     Concurrent RS threads times (s)
```

```
            0.00     0.00
     Concurrent sampling threads times (s)
            0.00

    Current rem set statistics
     Total per region rem sets sizes = 154K. Max = 5K.
           10K (  7.0%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
          143K ( 93.0%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      Static structures = 7K, free_lists = 0K.
       0 occupied cards represented.
             0 (  0.0%) entries by 2 Young regions
             0 (  0.0%) entries by 0 Humonguous regions
             0 (  0.0%) entries by 28 Free regions
             0 (  0.0%) entries by 0 Old regions
       Region with largest rem set = 29:(E)[0x00000000c7c00000,0x00000000c7e00000,
0x00000000c7e00000], size = 5K, occupied = 0K.
     Total heap region code root sets sizes = 1K.  Max = 0K.
            0K ( 56.9%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
            0K ( 43.1%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
      21 code roots represented.
            21 (100.0%) elements by 2 Young regions
             0 (  0.0%) elements by 0 Humonguous regions
             0 (  0.0%) elements by 28 Free regions
             0 (  0.0%) elements by 0 Old regions
       Region with largest amount of code roots = 29:(E)[0x00000000c7c00000,
0x00000000c7e00000,0x00000000c7e00000], size = 0K, num_elems = 0.
```

20. –XX:+HeapDumpBeforeFullGC/–XX:+HeapDumpAfterFullGC

这个选项启用之后，在 Full GC 开始之前有一个 hprof 文件会被创建。建议这个选项和-XX:+HeapDumpAfterFullGC 一起使用，可以通过对 Full GC 发生前后的 Java 堆内存进行对比，找出内存泄漏和其他问题。设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintGCApplicationStoppedTime -XX:G1HeapRegionSize=2M -XX:G1HeapWastePercent= 5 -XX:G1MixedGCCountTarget=10 -XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo -XX:G1ReservePercent=10 -XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1SummarizeRSetStats -XX:+G1TraceConcRefinement -XX:+G1UseAdaptiveConcRefinement -XX:

GCTimeRatio=10 -XX:+HeapDumpBeforeFullGC。运行输出如代码清单 3-27 所示。

代码清单 3-27　-XX: +HeapDumpBeforeFullGC 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1893480k free), swap 7843228k(5628592k free)
   CommandLine flags: -XX:G1HeapRegionSize=2097152 -XX:G1HeapWastePercent=5
-XX:G1MixedGCCountTarget=10 -XX:+G1PrintRegionLivenessInfo
-XX:G1ReservePercent=10 -XX:+G1SummarizeRSetStats
-XX:G1SummarizeRSetStatsPeriod=10 -XX:+G1TraceConcRefinement
-XX:+G1UseAdaptiveConcRefinement -XX:GCTimeRatio=10 -XX:+HeapDumpAfterFullGC
-XX:+HeapDumpBeforeFullGC -XX:InitialHeapSize=62760512
-XX:MaxHeapSize=1004168192 -XX:+PrintGC -XX:+PrintGCApplicationStoppedTime
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UnlockDiagnosticVMOptions
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation
   1.083: Total time for which application threads were stopped: 0.0001209
seconds, Stopping threads took: 0.0000313 seconds
   3.083: Total time for which application threads were stopped: 0.0000741
seconds, Stopping threads took: 0.0000300 seconds
   4.383: Total time for which application threads were stopped: 0.0001603
seconds, Stopping threads took: 0.0000334 seconds
   5.365: Total time for which application threads were stopped: 0.0000797
seconds, Stopping threads took: 0.0000330 seconds
   G1-Refine-stop
   G1-Refine-stop
   G1-Refine-stop
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c44000f0, 0x0000000100000000)
     region size 2048K, 2 young (4096K), 0 survivors (0K)
    Metaspace       used 3973K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
    Cumulative RS summary

    Recent concurrent refinement statistics
     Processed 0 cards
     Of 0 completed buffers:
            0 (  0.0%) by concurrent RS threads.
            0 (  0.0%) by mutator threads.
```

```
    Did 0 coarsenings.
    Concurrent RS threads times (s)
           0.00     0.00
    Concurrent sampling threads times (s)
           0.00

   Current rem set statistics
    Total per region rem sets sizes = 154K. Max = 5K.
           10K (  7.1%) by 2 Young regions
            0K (  0.0%) by 0 Humonguous regions
          143K ( 92.9%) by 28 Free regions
            0K (  0.0%) by 0 Old regions
     Static structures = 7K, free_lists = 0K.
      0 occupied cards represented.
             0 (  0.0%) entries by 2 Young regions
             0 (  0.0%) entries by 0 Humonguous regions
             0 (  0.0%) entries by 28 Free regions
             0 (  0.0%) entries by 0 Old regions
      Region with largest rem set = 29:(E)[0x00000000c7c00000,0x00000000c7e00000,
0x00000000c7e00000], size = 5K, occupied = 0K.
    Total heap region code root sets sizes = 1K.  Max = 0K.
           0K ( 61.4%) by 2 Young regions
           0K (  0.0%) by 0 Humonguous regions
           0K ( 38.6%) by 28 Free regions
           0K (  0.0%) by 0 Old regions
     26 code roots represented.
           26 (100.0%) elements by 2 Young regions
            0 (  0.0%) elements by 0 Humonguous regions
            0 (  0.0%) elements by 28 Free regions
            0 (  0.0%) elements by 0 Old regions
      Region with largest amount of code roots = 29:(E)[0x00000000c7c00000,
0x00000000c7e00000,0x00000000c7e00000], size = 0K, num_elems = 0.
```

21. -XX:InitiatingHeapOccypancyPercent

该选项的默认值是 45，表示 G1 GC 并行循环初始设置的堆大小值，这个值决定了一个并行循环是不是要开始执行。它的逻辑是在一次 GC 完成后，比较老年代占用的空间和整个 Java 堆之间的比例。如果大于这个值，则预约下一次 GC 开始一个并行循环回收垃圾，从初始标记阶段开始。这个值越小，GC 越频繁，反之，值越大，可以让应用程序执行时间更长。不过在内存消耗很快的情况下，我认为早运行并行循环比晚运行要好，看病要趁早。

我们设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:InitiatingHeapOccupancyPercent=5，运行输出如代码清单 3-28 所示。

代码清单 3-28　–XX: InitiatingHeapOccupancyPercent 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1882348k free), swap 7843228k(5605408k free)
   CommandLine flags: -XX:InitialHeapSize=62760512
-XX:InitiatingHeapOccupancyPercent=5 -XX:MaxHeapSize=1004168192 -XX:+PrintGC
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   Heap
    garbage-first heap   total 61440K, used 3072K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

22. –XX:+ UseStringDeduplication

该选项启动 Java String 对象的去重工作。JDK8u20 开始引入该选项，默认为不启用。我们知道一个判断 Java String 对象值是否一样的语句“String1.equals(String2)=true”，如果开启了该选项，并且如果两个对象包含相同的内容，即返回“true”，则两个 String 对象只会共享一个字符数组。这个选项是 G1 GC 独有的，也可以和其他 GC 一起使用。

延伸一点我们的知识面，一个去重对象的必备条件有如下三点。

- Java.lang.String 对象的一个实例。
- 这个对象在年轻代堆区间。
- 这个对象的年龄达到去重年龄代，或者这个对象已经在老年代堆区间并且对象年龄比去重年龄小。选项-XX:StringDeduplicationAgeThreshold 设置了这个年龄界限。

前面介绍过的可修改和不可修改字符串的处理方式有所不同，不可修改字符串默认就是去重的，在插入到 HotSpot VM 的 StringTable 时已经注明了是去重的，这样就避免了 HotSpot 服务器 JIT 编译优化措施。

设置 VM 选项：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+UseStringDeduplication。运行输出如代码清单 3-29 所示。

代码清单 3-29　-XX: InitiatingHeapOccupancyPercent 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1896116k free), swap 7843228k(5633992k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
-XX:-UseLargePagesIndividualAllocation -XX:+UseStringDeduplication
   Heap
    garbage-first heap   total 61440K, used 3072K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3953K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

23. -XX:StringDeduplicationAgeThreshold

这个选项是针对-XX:+ UseStringDeduplication 选项的，默认值是 3。它的意思是一个字符串对象的年龄超过设定的阈值，或者提升到 G1 GC 老年代 Region 之后，就会成为字符串去重的候选对象，去重操作只会有一次。

设置 VM 选项：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+UseStringDeduplication -XX:StringDeduplicationAgeThreshold=1。运行输出如代码清单 3-30 所示。

代码清单 3-30　-XX:StringDeduplicationAgeThreshold 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1574264k free), swap 7843228k(5222608k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:StringDeduplicationAgeThreshold=1 -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
-XX:+UseStringDeduplication
   Heap
    garbage-first heap   total 61440K, used 2048K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 3 young (3072K), 0 survivors (0K)
    Metaspace       used 3954K, capacity 4716K, committed 4864K, reserved 1056768K
```

```
    class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

24. –XX:+PrintStringDeduplicationStatistics

这个选项挺有用的，能够帮助我们通过读取输出的统计资料来了解是否字符串去重后节约了大量的堆内存空间，默认是关闭的，就是说不会输出字符串去重的统计资料。

设置 VM 选项：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+UseStringDeduplication -XX:StringDeduplicationAgeThreshold=1 -XX:+PrintStringDeduplicationStatistics，运行日志输出如代码清单 3-31 所示。

代码清单 3–31　–XX: +PrintStringDeduplicationStatistics 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1551852k free), swap 7843228k(5212456k free)
   CommandLine flags: -XX:InitialHeapSize=62760512 -XX:MaxHeapSize=1004168192
-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+PrintStringDeduplicationStatistics
-XX:StringDeduplicationAgeThreshold=1 -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
-XX:+UseStringDeduplication
   Heap
    garbage-first heap   total 61440K, used 3072K [0x00000000c4200000,
0x00000000c43001e0, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3948K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

25. –XX:+G1UseAdaptiveIHOP

JDK9 提供的新的选项。这个选项的作用是通过动态调节标记阶段开始的时间，以达到提升应用程序吞吐量的目标，主要通过尽可能迟地触发标记循环方式来避免消耗老年代空间。

这个选项的值在 VM 刚开始启动时和-XX:InitiatingHeapOccupancyPercent 的值一样，如果出现标记循环阶段内存不够用，则它会自动调节大小，确保标记循环启用更多的堆内存。

注意，-XX:+G1UseAdaptiveIHOP 这个选项会在 JDK9 里默认启用，即-XX:InitiatingHeapOccupancyPercent 和-XX:+G1UseAdaptiveIHOP 在 JDK9 之后只需要启用一个就可以了。

JDK8 环境下运行该选项会输出：“Unrecognized VM option 'G1UseAdaptiveIHOP'”。

26. –XX:MaxGCPauseMills

这个选项比较重要。它设置了 G1 的目标停顿时间，单位是 ms，默认值为 200ms。这个值是一个目标时间，而不是最大停顿时间。G1 GC 尽最大努力确保年轻代的回收时间可以控制在这个目标停顿时间范围里面，在 G1 GC 使用过程中，这个选项和-Xms、-Xmx 两个选项一起使用，它们三个也最好在 JVM 启动时就一起配置好。

设置 VM 选项为：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:MaxGCPauseMillis=10 -Xms200m -Xmx1024m，运行输出日志如代码清单 3-32 所示。

代码清单 3–32 –XX:MaxGCPauseMills 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1405972k free), swap 7843228k(4985964k free)
   CommandLine flags: -XX:InitialHeapSize=209715200 -XX:MaxGCPauseMillis=10
-XX:MaxHeapSize=1073741824 -XX:+PrintGC -XX:+PrintGCDetails
-XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
-XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   Heap
    garbage-first heap   total 204800K, used 3072K [0x00000000c0000000,
0x00000000c0100640, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3968K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

27. –XX:MinHeapFreeRatio

这个选项设置堆内存里可以空闲的最小的内存空间大小，默认值为堆内存的 40%。当空闲堆内存大小小于这个设置的值时，我们需要判断-Xms 和-Xmx 这两个值的初始化设置值，如果-Xms 和-Xmx 不一样，那么我们就有机会扩展堆内存，否则就无法扩展。

尝试设置 VM 选项：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:MinHeapFreeRatio=99 -Xms200m -Xmx1024m。

这时候 JVM 会在初始化时报错：

```
MinHeapFreeRatio (99) must be less than or equal to MaxHeapFreeRatio (70)
```

这个错误告诉我们最小空闲内存必须小于堆内存，所以配置为 70%，运行日志输出如代码清单 3-33 所示。

代码清单 3-33　-XX: MinHeapFreeRatio 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1862044k free), swap 7843228k(5638644k free)
   CommandLine flags: -XX:InitialHeapSize=209715200
-XX:MaxHeapSize=1073741824 -XX:MinHeapFreeRatio=70 -XX:+PrintGC
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
-XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
   Heap
    garbage-first heap   total 204800K, used 3072K [0x00000000c0000000,
0x00000000c0100640, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3961K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

28. -XX:MaxHeapFreeRatio

这个选项设置最大空闲空间大小，默认值为堆内存的 70%。这个选项和上面那个最小堆内存空闲大小刚好相反，当大于这个空闲比率时，G1 GC 会自动减少堆内存大小。需要判断-Xms 和-Xmx 这两个值的初始化设置值，如果-Xms 和-Xmx 不一样，那么就有机会减小堆内存，否则就无法减小。

29. -XX:+PrintAdaptiveSizePolicy

这个选项决定是否开启堆内存大小变化的相应记录信息打印，即是否打印这些信息到 GC 日志里面。这个信息对于 Parallel GC 和 G1 GC 都很有用。设置 VM 选项：-XX:+PrintGCDetails -verbose:gc -Xloggc:gc.log -XX:+UseG1GC -XX:+PrintAdaptiveSizePolicy -XX:MinHeapFreeRatio=70 -XX:MaxHeapFreeRatio=99 -Xms200m -Xmx1024m，运行日志输出如代码清单 3-34 所示。

代码清单 3-34　-XX: +PrintAdaptiveSizePolicy 运行输出

```
   Java HotSpot(TM) 64-Bit Server VM (25.101-b13) for windows-amd64 JRE
(1.8.0_101-b13), built on Jun 22 2016 01:21:29 by "Java_re" with MS VC++ 10.0
(VS2010)
   Memory: 4k page, physical 3922532k(1624280k free), swap 7843228k(5310472k free)
   CommandLine flags: -XX:InitialHeapSize=209715200 -XX:MaxHeapFreeRatio=99
-XX:MaxHeapSize=1073741824 -XX:MinHeapFreeRatio=70
-XX:+PrintAdaptiveSizePolicy -XX:+PrintGC -XX:+PrintGCDetails
-XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops
```

```
-XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
    0.021: [G1Ergonomics (Heap Sizing) expand the heap, requested expansion
amount: 209715200 bytes, attempted expansion amount: 209715200 bytes]
   Heap
    garbage-first heap   total 204800K, used 3072K [0x00000000c0000000,
0x00000000c0100640, 0x0000000100000000)
     region size 1024K, 4 young (4096K), 0 survivors (0K)
    Metaspace       used 3954K, capacity 4716K, committed 4864K, reserved 1056768K
     class space    used 455K, capacity 468K, committed 512K, reserved 1048576K
```

30. –XX:+ResizePLAB

GC 使用的本地线程分配缓存块采用动态值还是静态值进行设置是由这个选项决定的，它默认是开启的，这个设置对应的是 GC 在提升对象时是否会调整 PLAB 的大小。

这个选项大家还是慎用，据说会出现性能问题，启用后可能会增加 GC 的停顿时间。当应用开启的线程较多时，最好使用-XX:-ResizePLAB 来关闭 PLAB()的大小调整，以避免大量的线程通信所导致的性能下降。

31. –XX:+ResizeTLAB

Java 应用线程使用的本地线程分配缓存块采用动态值还是静态值进行设置是由这个选项决定的，它默认是开启的，即 TLAB 值会被动态调整。

我们通过示例看一下 TLAB 的作用，程序如代码清单 3-35 所示。

代码清单 3–35　TLAB 使用示例

```
public class TestUseTLAB {
    public static void allocdemo(){
        byte[] by = new byte[2];
        by[0] = 1;
    }

    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        for(int i = 0;i<10000000;i++){
            allocdemo();
        }
        long endTime = System.currentTimeMillis();
        System.out.println(endTime-startTime);
    }
}
```

我们可以看到在默认 VM 情况下（默认开启 TLAB），耗时 14ms，在使用-UseTLAB 关闭 TLAB 后，耗时 24ms。

可以看到，TLAB 是否启用，对于对象分配的影响是很大的。由于 TLAB 空间一般不会太大，因此大对象无法在 TLAB 上进行分配，总是会直接分配在堆上。因为 TLAB 空间比较小，因此很容易被装满。如果开启了 ResizTLAB，TLAB 会在运行时不断调整，使系统的运行状态达到最优。

和 PLAB 值动态调整不同的是，这个选项开启后会较大地提升应用程序性能，因为 TLAB 的竞争、切换减少了。所以我们默认还是开启它吧。

32. –XX:+ClassUnloadingWithConcurrentMark

这个选项开启在 G1 GC 并行循环阶段卸载类，尤其是在老年代的并行回收阶段，默认是开启的。这个选项开启后会在并行循环的重标记阶段卸载 JVM 没有用到的类，这些工作也可以放在 Full GC 里面去做，但是提前做了有很大的好处。但因为开启它意味着重标记阶段的 GC 停顿时间会拉长，这时候我们就要判断性价比了，如果 GC 停顿时间比我们设置的最大 GC 停顿目标时间还长，并且需要卸载的类也不多，那还是关闭这个选项吧。

33. –XX:+ClassUnloading

默认值是 Ture，决定了 JVM 是否会卸载所有无用的类，如果关闭了这个选项，无论是并行回收循环，还是 Full GC，都不会再卸载这些类了，所以需谨慎关闭。

34. –XX:+UnlockDiagnosticVMOptions

这个选项决定是否开启诊断选项，默认值是 False，即不开启。

在 GC 里面有一些选项称之为诊断选项（Diagnostic Options），通过-XX:+PrintFlagsFinal 和-XX:+UnlockDiagnosticVMOptions 这两个选项组合起来运行，就可以输出并查看这些选项，输出如代码清单 3-36 所示。

代码清单 3–36　诊断选项设置列表输出

```
[Global flags]
    uintx AdaptiveSizeDecrementScaleFactor          = 4          {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10         {product}
    uintx AdaptiveSizePausePolicy                   = 0          {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50         {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20
```

```
    bool CrashOnOutOfMemoryError                    = false
    intx hashCode                                   = 5          {product}
```

35. –XX:+UnlockExperimentalVMOptions

除了之前说的诊断选项以外，JVM 还有一些叫作试验选项（Experimental Options），这些选项也需要通过-XX:+UnlockExperimentalVMOptions 这个选项开启，默认是关闭的。

和诊断选项一样，也可以和-XX:+PrintFlagsFinal 选项联合使用，即-XX:+PrintFlagsFinal 和-XX:+UnlockExperimentalVMOptions 这两个选项联合使用时可以输出日志，输出的日志已经包含在了前一个选项-XX:+UnlockDiagnosticVMOptions 的运行输出里，这里就不再重复。

总的来说，这些试验选项对整体应用性能可能会有些好处，但是它们并没有经历完整的测试环节，所以称为试验选项。

举一个例子，比如-XX:G1NewSizePercent，这个选项控制 G1 GC 的最小堆大小，可以通过这个选项来减小 Java 堆区里年轻代的大小，默认是堆大小的 5%。假设应用场景需要很短的 GC 中断时间，这时候可以设置该选项为 1%或 2%，这样年轻代空间就减小了。为了启用这个选项，需要和-XX:+UnlockExperimentalVMOptions 联合使用，如-XX:+UnlockExperimentalVMOptions –XX:G1NewSizePercent=2。

36. –XX:+UnlockCommercialFeatures

这个选项判断是否使用 Oracle 特有的特性，默认是关闭的。

有一些属性是 Oracle 公司针对 Oracle 的 Java 运行时独有的，没有被包含在 OpenJDK 里面。举个例子，比如说 Oracle 的监控和管理工具 Java Mission Control，它有一个特性叫作 Java Flight Recorder，这个特性作为 Java Mission Control 的一部分，属于事件回收框架，可以被用来显示应用程序和 JVM 的底层信息。

3.3 本章小结

本章首先提出了一个范例程序，然后从 Print GC 日志选项开始，逐一介绍了几十个 JVM 命令行选项，一部分是G1 GC独有的选项，通过这些选项的逐一讲解，让大家能够动手使用G1 GC，深入的知识会在第 4、第 5 章详细讲解。

4 第4章 深入G1 GC

G1 GC采用递增、并行运算、独占式运算的特征方式，并采用拷贝技术实现自身的压缩目标。同时，通过并行的多级标记方式缩短各层级（标记、重标记、清除等阶段）的停顿时间。我们经常可以看到美国大片里男主角轻松地避开安检，一人独闯敌人老巢的场景。我想说的是，G1 GC的多层级、无间隔排查设计方案，让这种场景只能发生在电视屏幕上了。

本章是全书的概念层面核心章节，主要介绍和解决以下问题。

- 深入学习G1 GC的各种基本概念。
- 深入学习G1 GC的设计理念。
- 深入了解G1 GC的各类特性。
- 为第5章节针对G1 GC的性能优化方案做知识准备。

4.1 G1 GC 概念简述

4.1.1 背景知识

作为 HotSpot VM 家族的新成员，G1 GC 的全称是 Garbage First GC，为什么会叫这个名字呢？因为 G1 GC 是一个压缩收集器，它基于回收最大量的垃圾原理进行设计。G1 GC 利用递增、并行、独占暂停这些属性，通过拷贝的方式完成压缩目标。此外，它也借助并行、多阶段并行标记这些方式来帮助减少标记、重标记、清除暂停的停顿时间，让停顿时间最小化也是它的设计目标之一。

G1 回收器是在 JDK1.7 中正式投入使用的全新的垃圾回收器，从长期目标来看，它是为了取代 CMS[1]回收器。G1 回收器拥有独特的垃圾回收策略，这和之前提到的回收器截然不同。从分代上看，G1 依然属于分代型垃圾回收器，它会区分年轻代和老年代，年轻代依然有 Eden 区和 Survivor 区，但从堆的结构上看，它并不要求整个 Eden 区、年轻代或者老年代包含的 Region 区在物理上都是连续的。综合来说，G1 使用了全新的分区算法，其特点如下所示。

- 并行性：G1 在回收期间，可以有多个 GC 线程同时工作，可以有效利用多核的计算能力。
- 并发性：G1 拥有与应用程序交替执行的能力，部分工作可以和应用程序同时执行，因此，一般来说，不会在整个回收阶段发生完全阻塞应用程序的情况。
- 分代 GC：G1 依然是一个分代收集器，但是和之前的各类回收器不同，它同时兼顾了年轻代和老年代。对比其他回收器，它们或者工作在年轻代，或者工作在老年代。
- 空间整理：G1 在回收过程中，会进行适当的对象移动，不像 CMS 那样只是简单地标记清理对象。在若干次 GC 后，CMS 必须进行一次碎片整理。而 G1 不同，它每次回收都会有效地复制对象，减少空间碎片，进而提升内部循环速度。
- 可预见性：由于分区的原因，G1 可以只选取部分区域进行内存回收，这样缩小了回收的范围，因此对于全局停顿情况的发生也能得到较好的控制。

随着 G1 GC 的出现，GC 从传统的连续堆内存布局逐渐走向了不连续内存块布局，这是通过引入 Region 概念实现[2]的，也就是说，由一堆不连续的 Region 组成了堆内存。其实也不能说是不连续的，只是它从传统的物理连续逐渐改变为逻辑上的连续，这是通过 Region 的动态分配

---

[1] 即 Concurrent-Mark-Sweep。

[2] 事实上很多软件产品都有这样的经历，HBase 的 Region 概念、关系型数据库的 Partition 概念，都是类似的，也是必然会出现的解决方案，分而治之、灵活调配。

方式实现的，可以把一个 Region 分配给 Eden、Survivor、老年代、大对象区间、空闲区间等区间的任意一个，而不是固定它的作用，因为越是固定，越是呆板。

4.1.2　G1 的垃圾回收机制

通过市场的力量，可以不断地淘汰旧的行业，把有限的资源让给那些竞争力更强、利润率更高的企业，类似地，硅谷也在不断淘汰过时的人员，从全世界吸收新鲜血液。经过半个多世纪的发展，在硅谷地区便形成了只有卓越才能生存的文化。本着这样的理念，GC 承担了淘汰垃圾、保存优良资产的任务。

G1 GC 在回收暂停阶段会回收最大量的堆内区间（Region），这是它的设计目标，通过回收区间达到回收垃圾的目的。这里只有一个例外情况，这个例外发生在并行标记阶段的清除（Cleanup）步骤，如果 G1 GC 在清除步骤发现所有的区间都是由可回收垃圾组成的，那么它会立即回收这些区间，并且将这些区间插入到一个基于 LinkedList 实现的空闲区间队列里，以待后用。因此，释放这些区间并不需要等待下一个垃圾回收的中断，它是实时执行的，即清除阶段起到了最后一道把控作用。这是 G1 GC 和之前的几代 GC 的一大差别。

G1 GC 的垃圾回收循环由三个主要类型组成：年轻代循环[1]、多步骤并行标记循环[2]、混合收集循环[3]。当然，单线程、独占式、高强度的 Full GC 还是继续存在的，它针对 GC 的评估失败[4]提供了一种失败保护机制，即强力回收。

在年轻代回收期，G1 GC 暂停应用程序线程，然后从年轻代区间移动存活对象到幸存者区间或者老年代区间，也有可能是两个区间都会涉及。对于一个混合回收期，G1 GC 从老年代区间移动存活对象到空闲区间，这些空闲区间也就成为了老年代的一部分。

4.1.3　G1 的区间设计灵感

为了加快 GC 的回收速度，HotSpot 的历代 GC 都有自己的不同的设计方案，读者可以看前三个章节，这里直接介绍 G1 GC 的 Region 设计方案。区间概念在软件设计、架构领域并不是一个新名词，关系型数据库、列式数据库最先使用这个概念提升数据存取速度，软件架构设计时也广泛使用这样的分区概念加快数据交换、计算。

---

[1] 即 Young Collection Cycle。

[2] 即 Multistage Concurrent Marking Cycle。

[3] 即 Mixed Collection Cycle。

[4] 即 Evacuaion failure。

为什么会有区间这个设计想法？大家一定看过电视剧《大宅门》吧？大宅门所描述的北京知名医术世家白家是这部电视剧的主角，白家有三兄弟，没有分家之前，由老爷子一手掌管全家，老爷子看似是个精明人，实质是个糊涂的人，否则也不会弄得后来白家家破人散。白家的三兄弟在没有分家之前，老大一家很老实，老二很懦弱，性格像女人，虽然肚子里明白道理，但是不敢出来做主。老三年轻时是混蛋一个，每次外出采购药材都要私吞家里的银两，造成账目混乱。老大为了家庭和睦，一直在私下倒贴银两，让老爷子能够看到一本正常的账目。这样的一家子聚在一起，家庭内部迟早会出现问题，倒不如分家，你也不用算计家里的钱了，分给你的钱有本事守住，没本事就一直拮据下去吧。这就是最原始的分区（Region）概念。

回到技术，看看 HBase 的 RegionServer 设计方式。在 HBase 内部，所有的用户数据以及元数据的请求，在经过 Region 的定位后，最终会落在 RegionServer 上，并由 RegionServer 实现数据的读写操作。RegionServer 是 HBase 集群运行在每个工作节点上的服务，它是整个 HBase 系统的关键所在，一方面它维护了 Region 的状态，提供了对于 Region 的管理和服务；另一方面，它与 Master 交互，上传 Region 的负载信息，参与 Master 的分布式协调管理，如图 4-1 所示。

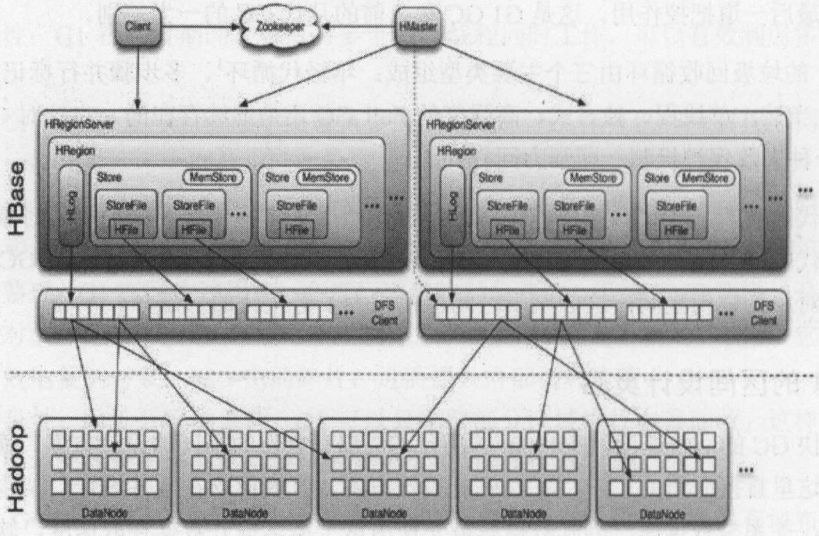


图 4-1　HBase 系统架构图

HRegionServer 与 HMaster 以及 Client 之间采用 RPC 协议进行通信。HRegionServer 向 HMaster 定期汇报节点的负载状况，包括 RS 内存的使用状态、在线状态的 Region 等信息，在该过程中 HRegionServer 扮演了 RPC 客户端的角色，而 HMaster 扮演了 RPC 服务器端的角色。HRegionServer 内置的 RpcServer 实现了数据更新、读取、删除的操作，以及 Region 涉及的 Flush、

Compaction、Open、Close、Load 等功能性操作。

Region 是 HBase 数据存储和管理的基本单位。HBase 使用 RowKey 将表水平切割成多个 HRegion，从 HMaster 的角度，每个 HRegion 都记录了它的 StartKey 和 EndKey（第一个 HRegion 的 StartKey 为空，最后一个 HRegion 的 EndKey 为空），由于 RowKey 是排序的，因而 Client 可以通过 HMaster 快速地定位每个 RowKey 在哪个 HRegion 中。HRegion 由 HMaster 分配到相应的 HRegionServer 中，然后由 HRegionServer 负责 HRegion 的启动和管理以及和 Client 的通信，并负责数据的读取（使用 HDFS）。每个 HRegionServer 可以同时管理 1000 个左右的 HRegion。RegionServer 管理拓扑图如图 4-2 所示。

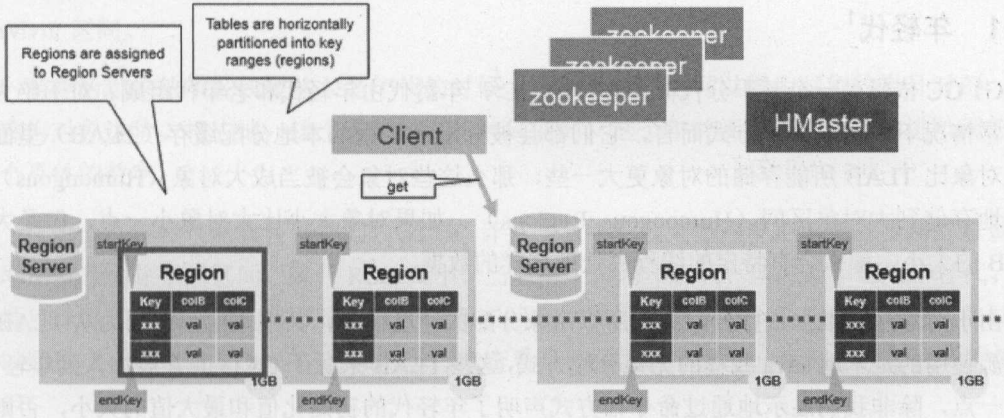


图 4-2　RegionServer 管理拓扑图

再来看看软件系统架构方面的分区设计。以任务调度为例，假设有一个中心调度服务，那么当数据量不断增多时，这个中心调度服务一定会遇到性能瓶颈，因为所有的请求都会最终指向它。为了解决这个性能瓶颈，可以将任务调度拆分为多个服务，即这多个服务都可以处理任务调度工作，那么问题来了，每个任务调度服务处理的源数据是否需要完全一致呢？根据华为公司发布的发明专利，它们对于每一个任务调度服务有数据来源区分的操作，即按照任务调度数量对源数据进行划分，比如 3 个任务调度服务，那么源数据按照行号对 3 取余的方式划分，如果运行了一段时间之后，任务调度服务出现了数量上的增减，那么这个取余划分需要重新进行，要按照这个时候的任务调度数量重新划分区间。

回到 G1。在 G1 中，堆被平均分成若干个大小相等的区域（Region）。每个 Region 都有一个关联的 Remembered Set（简称 RS），RS 的数据结构是 Hash 表，里面的数据是 Card Table（堆中每 512byte 映射在 card table 1byte）。简单地说，RS 里面存在的是 Region 中存活对象的指针。

当 Region 中数据发生变化时，首先反映到 Card Table 中的一个或多个 Card 上，RS 通过扫描内部的 Card Table 得知 Region 中内存使用情况和存活对象。在使用 Region 过程中，如果 Region 被填满了，分配内存的线程会重新选择一个新的 Region，空闲 Region 被组织到一个基于链表的数据结构（LinkedList）里面，这样可以快速找到新的 Region。

4.2 G1 GC 分代管理

G1 GC 的收集过程涵盖 4 个阶段，即年轻代 GC、并发标记周期、混合收集、Full GC。

4.2.1 年轻代[1]

G1 GC 依然是一个基于分代概念设计的 GC[2]，年龄代由年轻代和老年代组成。对于绝大多数正常情况下的对象分配方式而言，它们都会被分配在线程的本地分配缓存（TLAB）里面。如果对象比 TLAB 所能存储的对象更大一些，那么这些对象会被当成大对象（Humongous），相应地存储到大对象区间（Humongous Regions）。如果对象大小比大对象小一点，但是大于 TLAB 的大小，那么会有特定的线程去处理这样的数据。

由于对象是在堆上进行分配的，所以对象分配的速度很快，其根本原因是因为从 TLAB 调动资源使用的是本地 Java 线程的无锁分配方式，这些 TLAB 来源于 G1 区间并被加入到年轻代。注意一点，除非我们显示地通过命令行方式声明了年轻代的初始化值和最大值的大小，否则一般来说[3]，初始化值默认是整个 Java 堆大小的 5%（通过选项-XX:G1NewSizePercent 设置），最大值默认是整个 Java 堆大小的 60%（通过选项-XX:G1MaxNewSizePercent 设置）。每次 GC 的停顿目标时间由选项-XX:MaxGCPauseMills 设置，注意，默认值是 200ms。如果用户设置了-Xmn 或者对应的年轻代大小（-XX:NewRatio），那么 G1 GC 会自动忽略该选项设置，进而忽略年轻代大小。

在年轻代最大空间没有到达前，G1 GC 会根据 Java 应用对象的分配速率，从空闲区间里面挑选出区间加入年轻代。堆区间大小在 JVM 启动时设置，大小从 1MB 到 32MB 不等，且必须是 2 的倍数，即 1MB、2MB、4MB、8MB、16MB 或 32MB。JVM 最多可以容纳大概 2048 个区间，也就是说，每个区间的大小为堆大小除以 2048。每个区间大小通过选项-XX:G1HeapRegionSize

---

[1] 即 The Young Generation。

[2] Parallel GC 和 CMS GC 也是的，所以说是依然。

[3] JDK 8u45 之后。

设置。这个过程可以理解为是一个不断增加的，且每次增加的是固定大小的区间，直到到达最大限定值为止的过程。

4.2.2　年轻代回收暂停[1]

年轻代是由 Eden 和 Survivor 两个区间组成的，那么当 Eden 区间分配内存失败，也可以说是当内存完全占满的时候，一次年轻代回收就被触发了。这次触发对于 GC 来说，它的工作是释放一些内存，属于一次轻量级的回收操作。首先，GC 需要把所有的存活对象从 Eden 区间移动到 Survivor 区间，这就是所谓的拷贝到幸存者区间的动作，这个操作类似于交换算法操作。任何一个年轻代回收都会提升整个年轻代的存活对象（由 Eden 和 Survivor 区间组成）到新的 Survivor 区间。

当存活对象从年轻代被移入老年代的时候，这一时刻我们可以称为对象的提升时刻，也可以称为对象进阶。相应地，这个进阶是有年龄阈值的，这个年龄阈值就是所谓的进阶阈值，是一个具体的数字。

在每一次年轻代回收暂停期间，G1 GC 计算当前年轻代大小需要扩展或者压缩的总量，例如增加或者删除空闲区间、统计 RSet 大小、当前最大可用年轻代、当前最小可用年轻代、设置停顿目标等。因此，我们可以认为这个过程在回收停顿结束后是一个重新调整年轻代的过程。可以通过-XX:+PrintGCDetails 选项的运行来查看具体数据。下面举一个例子，如代码清单 4-1 所示。

代码清单 4-1　年轻代回收示例

```
   15.002:[GC pause (G1 Evacuation Pause) (young),0.014821 secs]
     [Parallel Time: 9.5ms, GC Workers:8]
   <snip>
     [Code Root Fixup:  0.1 ms]
     [Code Root Purge: 0.0 ms]
     [Other: 4.9ms]
   <snip>
   [Eden: 695.0M(695.0M)->0.0B(1038.0M) Survivors:10.0m->13.0M
Heap:741.5M(1175.0M)->54.0M(1175.0M)]
   [Time:user=0.08 sys=0.00,real=0.01 secs]
```

从示例 4-1 可以看出，年轻代的总大小可以通过 Eden 区加上 Survivor 区来计算，即最后一

[1] 即 A Young Collection Pause。

行 1038.0MB+13.0MB，一共有 1051MB。

在整个年轻代回收过程中，幸存下来的对象都有自己的相关信息，我们姑且称为“Age Info”。我们会把年龄相关的信息加入到一个年龄表里面去，这个表是一个虚拟表。GC 维持了年龄信息，这些相关信息是为了记录幸存者区间的存活对象何时提升到老年区、提升的相关阈值等，这些记录为提升阶段做准备。此外，也为混合回收阶段、清洗阶段或者 Full GC 提供数据支撑。基于这张年龄表，幸存区大小、幸存者存活容量（由选项-XX:TargetSurvivorRatio 和-XX:MaxTenuringThreshold 来决定），JVM 动态地为所有存活对象设置提升（回收）阈值，这两个选项的默认值分别为 50%和 15%。一旦对象突破这个阈值，它们就可以被提升到老年代区间。当这些幸存对象在老年代死亡后，它们的空间就可以被混合回收阶段、清洗阶段、Full GC 阶段释放。

4.2.3 大对象区间[1]

一个封闭系统总是朝着熵增加的方向变化的，即从有序变为无序，只有从外界引入熵，才能变回到更有序的状态。因此一个公司、组织，如果引入多元文化，就会变得更好，这就是“他山之石，可以攻玉”的道理。反之，如果一个组织内只有单一文化，近亲繁殖，道路便会越走越窄。正是因为可以包容多元化，硅谷才能够不断进步。类似于这个道理解释的原理，G1 GC 引入了大对象区间，用于优化现有的 Java 对象交换、存储方式。

在 G1 GC 内部，我们把一个回收区间称为一个 Region，即前文里一直称呼的区间。通过-XX:G1HeapRegionSize 选项来设置一个区间可以有多大的空间，同时，堆区间大小也决定了什么对象可以被认定为“大对象”。大对象通常是很大的对象，占据了超过 G1 GC 所有区间的 50%。大对象没有按照年轻代的回收方式把对象放入老年区，而是放置在老年区以外的大对象区间里面，即单独分离出来形成一片新的区域，独立管理。

图 4-3 描述的是一个连续的 Java 堆区，该堆区由年轻代区间、老年代区间和大对象区间这三个不同类型的 G1 区间组成。

[1] 即 Humongous Regions。

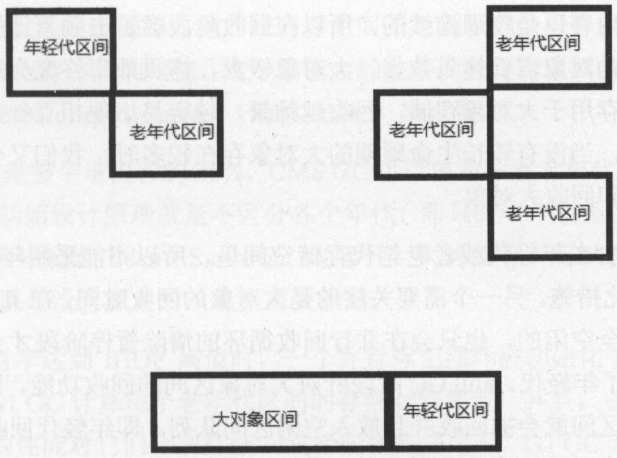


图 4-3　混合区间类型组成的 Java 堆区

从图 4-3 可以知道，每一个年轻代区间和老年代区间都是一个回收单元，而大对象区间覆盖了两个回收单元，并且这两个回收单元是连续的堆区间。让我们再通过图 4-4 深入地了解大对象区间。图 4-4 中，大对象 1 穿越了两个连续的区间，大约占了两个区间的 3/4，开始阶段叫作大对象开始（StartsHumogous），当它跨越到第二个连续区间的 1/2 个大区间后对象成为连续的大对象（ContinuesHumongous）。大对象 2 跨越了连续的 3 个区间，其中完全占据了前两个区间，并占据了第三个区间的 1/4，这样第一个区间就是大对象开始区间，第二个就是连续的大对象区间，第三个也是连续的大对象区间。对于大对象 3 而言，它只完全占据了一个区间，那么这两个连续的区间都可以称为大对象开始区间。

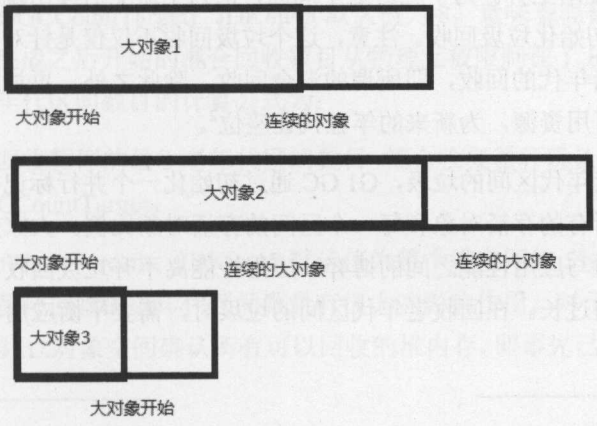


图 4-4　大对象对应大对象区间描述图

因为大对象在堆内存里是物理连续的，所以在回收阶段尝试去频繁地回收大对象，存在两个缺点，缺点一是移动对象需要拷贝数据，大对象较大，拷贝效率终究会成为瓶颈；缺点二是很难找到连续的堆内存用于大对象存储，回收越频繁，越容易出现相互独立的区间。随着应用程序业务逻辑的变化，当没有较长生命周期的大对象存在较多时，我们又会回到这个领域的研究，即如何优化分配和回收大对象。

大对象并没有包含在年轻代或者老年代存储空间里，所以不能采用年轻代针对 TLAB 和 PLAB 的分配或者优化措施。另一个需要关注的是大对象的回收时间，在 JDK8u40 之前的版本，即便大对象区间是完全空闲的，也只会在并行回收循环的清除暂停阶段才会回收大对象。而在 JDK8u40 之后，新增了年轻代、Full GC 阶段针对大对象区间的回收功能，只要大对象区间不再包含任何引用，这些区间就会被回收并且放入空闲区间队列。即年轻代回收、并行回收循环、Full GC，它们都会参与到大对象区间的回收工作。

如何定义大对象？

假设，当前每个区间的大小定义为 2MB，一个数组大小为 1MB。这个数组会被认定为大对象吗？是的，这是因为数组大小=数组内部对象的头大小+对象大小，即这个数组超过了 1MB，即超过了区间大小的 50%，符合大对象的定义要求，属于大对象。

4.2.4 混合回收暂停[1]

前面介绍了年轻代回收、大对象进入对应区间等操作方式，那么既然越来越多的对象从年轻代进入到了老年代，再加上进入单独区域的大对象区间，这时候老年代或者说整个 Java 堆区的负荷量/保有量也在逐渐上升。为了应对这个局面，也为了避免后续出现堆空间不足的情况发生，JVM 进程需要去初始化垃圾回收。注意，这个垃圾回收不仅仅是针对年轻代的，还需要能够额外增加一些针对老年代的回收，即所谓的混合回收。除此之外，再加上大对象区间也一起回收，以更多地回收可用资源，为新来的年轻代让座位[2]。

为了最大化回收老年代区间的垃圾，G1 GC 通过初始化一个并行标记循环来帮助标记对象的根节点，最终确认所有的存活对象和每一个区间的存活对象比例。大家知道垃圾回收过程实质上是一个垃圾回收器与应用性能之间的博弈，应用性能离不开垃圾回收，又不能让垃圾回收器堵塞应用程序的时间过长。在回收老年代区间的垃圾时，需要平衡应用与触发标记循环次数

---

[1] 即 A Mixed Collection Pause。

[2] 和社会退休体制是一样的，有内退、病退、正常退休等组成的混合退休方式，才能满足社会广泛需求。

之间的比例，我们的底线是确保不要抛出堆内存空间不足的缺陷给用户。因此，需要在 JVM 进程启动时设置一个堆内存占用的阈值，这个阈值会触发一次并行标记回收循环，通过 G1 GC 的选项-XX:InitiatingHeapOccupanyPercent（简称 IHOP）来设置。

IHOP 的默认值是整个堆内存的 45%。CMS GC 可以单独设置老年代相对于堆内存的阈值，但是由于 G1 GC 的初始设计原理就是不区分各个年代，即只有一个空闲区间池，在这个池子里面，区间可以被任意分配给 Eden、Survivor、老年代、大对象等，所以设置一个针对老年代的阈值对于 G1 GC 来说是没有意义的，也不需要设置。

当老年代的占有率达到 IHOP 阈值时，一个并行标记循环被初始化了，它启动了。在这个循环的末尾时刻，G1 GC 计算每个老年代区间的存活对象数量，并且，在清洗阶段，G1 GC 根据每个老年代区间的性能对它们分别打分[1]。这个阶段完成之后，G1 GC 开始一次混合回收。在混合回收暂停阶段，G1 GC 不仅仅回收年轻代区间，也会回收一些老年代区间，至于选择哪些老年代区间回收，这就取决于对这些区间的评估，标记有较多垃圾的老年代区间会被回收。[2]

一个单一的混合回收和年轻代回收暂停的工作方式类似，也是通过拷贝方式完成针对存活对象的压缩操作。唯一的区别是在混合回收阶段，回收集合也会和一些被认定为是可以做垃圾回收的老年代一起合作，即同时对年轻代和老年代区间进行回收。在一些选项的帮助下，可以为一次混合回收循环设定多次混合回收暂停。一次混合回收循环只有在标记或者超过 IHOP 阈值时才会触发启动，并且必须发生在一次并行标记循环之后。

上面提到了可以通过一些 JVM 选项来帮助决定一次混合回收循环内部的混合回收发生次数，这两个选项分别是-XX:G1MixedGCCountTarget 和-XX:G1HeapWastePercent。

- -XX:G1MixedGCCountTarget：JDK8u45 默认值为 8，意味着混合 GC 数目目标为 8 个，即标记循环完成之后开始的混合回收数目从物理上被限制在了 8 个。每一次混合回收暂停的最小老年代区间数目的计算公式为：

  每一次混合回收暂停的最小老年代区间数目=混合收集循环确认的候选老年代区间总数 / G1MixedGCCountTarget。

- -XX:G1HeapWastePercent：JDK8u45 默认值为整个堆空间的 5%，这个选项对于控制一次混合回收循环回收的老年代区间数量有很大的影响作用。对于每一次混合回收暂停，G1 GC 基于死亡对象空间确认所有可以回收的堆内存，即事先已经统计好了每个区间还

[1] 根据执行回收的效率、多少对象需要被回收、需要被回收的对象占据整个区间的百分比等计算出来。

[2] 年轻代会尽量多地回收，老年代需要等评估结果，挑选一部分可以回收的。

有多少存活对象，通过拷贝方式腾挪出区间，再把这些区间加入空闲区间。一旦 G1 GC 到达堆内存空闲阈值百分比限制时，G1 GC 停止初始化混合回收暂停，这样这个混合回收循环的作用就相应地结束了。设置堆空闲百分比有助于限制空闲的堆内存大小，也有助于提升混合回收循环的整体性能。

综上所述，一次混合回收循环内部的混合回收次数可以被每一次混合回收暂停阶段执行的最小老年代区间集合数量以及堆空闲百分比所控制。

4.2.5 回收集合及其重要性

任何一次垃圾回收都会释放 CSet 里面的所有区间。一个 CSet 由一系列的等待回收的区间所组成。在一次垃圾回收过程中，这些回收候选区间的存活对象会被整体评估，并且在回收结束后这些区间会被加入到空闲区间队列（LinkedList 队列）。在一次年轻代回收过程中，CSet 只会包含年轻代区间，而在一个混合回收过程中，CSet 会在年轻代区间基础上再包含一些老年代区间（前面一个章节具体介绍了入选条件），这就是新增的混合回收概念，不再对年轻代和老年代完全切分。

还是一样，G1 GC 提供了两个选项用于帮助选择进入 CSet 的候选老年代区间：-XX:G1MixedGCLiveThresholdPercent 和-XX:G1OldCSetRegionThresholdPercent。

- -XX:G1MixedGCLiveThresholdPercent：JDK8u45 默认值为一个 G1 GC 区间的 85%。这个值是一个存活对象的阈值，并且起到了从混合回收的 CSet 里排除一些老年代区间的作用，即可以理解为 G1 GC 限制 CSet 仅包含低于这个阈值（默认 85%）的老年代区间，这样可以减少垃圾回收过程中拷贝对象所消耗的时间。
- -XX:G1OldCSetRegionThresholdPercent：JDK8u45 默认值为整个 Java 堆区的 10%。这个值设置了可以被用于一次混合回收暂停所回收的最大老年代区间数量。这个阈值取决于 JVM 进程所能使用的 Java 堆的空闲空间。

4.2.6 RSet 及其重要性

基于年轻代、老年代、永久代（JDK8 淘汰）等分代理念、设计思维设计的垃圾回收器，它们采用将不同年龄段的对象存放在不同的堆内存区域的方式，可以让垃圾回收器高效地执行对象回收。垃圾回收器只需要扫描特定区域的对象，而不需要扫描整个堆内存区域，也不需要来回拷贝对象，减少了拷贝和更新引用的消耗。

为了满足这种回收器的设计思路，许多垃圾回收器针对每个年龄代维护了 RSet。一个 RSet

是一个数据结构，这个数据结构帮助维护和跟踪在它们单元内部的对象引用信息，在 G1 GC 里，这个单元就是区间（Region），也就是说，G1 GC 里每一个 RSet 对应的是一个区间内部的对象引用情况。有了 RSet，就不需要扫描整个堆内存了，当 G1 GC 执行 STW 独占回收（年轻代、混合代回收）时，只需要扫描每一个区间内部的 RSet 就可以了。因为所有 RSet 都保存在 CSet 里面，即 Region-RSet-CSet 这样的概念，所以一旦区间内部的存活对象被移除，RSet 里面保存的引用信息也会立即被更新。这样我们就能够理解 RSet 就是一张虚拟的对象引用表了，每个区间内部都有这么一张表存在，帮助对区间内部的对象存活情况、基本信息做有序高效的管理。

G1 GC 的年轻代回收或者混合回收阶段，由于年轻代被尽可能地设计为最大量的回收，这样的设计方式减少了对于 RSet 的依赖，即减弱了对于年轻代里面存储的跟踪引用信息的依赖程度，进而减弱了多余 RSet 的消耗。G1 GC 只在以下两个场景依赖 RSet。

- 老年代到年轻代的引用：G1 GC 维护了从老年代区间到年轻代区间的指针，这个指针保存在年轻代的 RSet 里面。
- 老年代到老年代的引用：G1 GC 维护了从老年代区间到老年代区间的指针，这个指针保存在老年代的 RSet 里面。

从图 4-5 中可以看到，有 1 个年轻代区间和 2 个老年代区间。区间 X 有一个从区间 Z 过来的引用，这个引用信息被保存在区间 X 的 RSet 里面。区间 Z 也有两个引用，一个来自于区间 X，另一个来自于区间 Y。区间 Z 的 RSet 只需要去保存从区间 Y 过来的引用，而不需要去刻意地记录从区间 X 过来的引用。由于年轻代会被完整地回收，所以这就解释了为什么区间 Z 不需要维护从年轻代区间 X 过来的引用了。

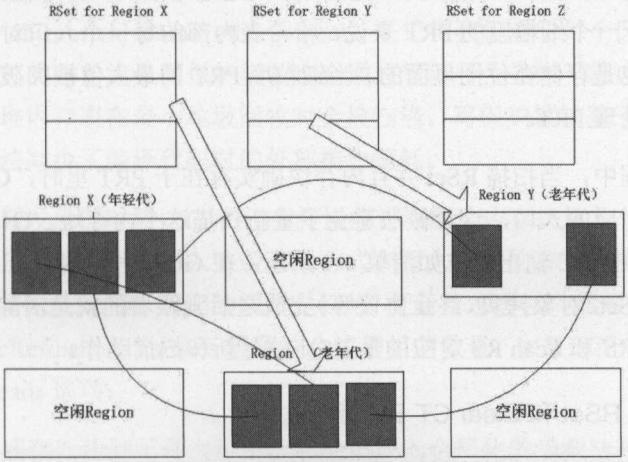


图 4-5　区间之间引用连接图

从图 4-5 可以知道每一个区间只会有一个 RSet。由于对于对象的引用是基于 Java 应用程序的需求的，所以有可能会出现 RSet 内部的“热点”，即一个区间出现很多次的引用更新，都出现在同一个位置的情况。

对于一个访问很频繁的区间来说，这样的方式会影响 RSet 的扫描时间。

注意，区间（Region）并不是最小单元，每个区间会被进一步划分为若干个块（Chunks）。在 G1 GC 区间里，最小的单元是一个 512 个字节的堆内存块（Card）。G1 GC 为每个区间设置了一个全局内存块表来帮助维护所有的堆内存块，如图 4-6 所示。

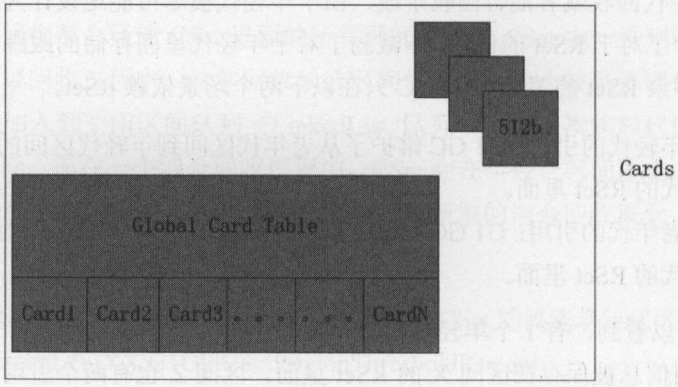


图 4-6　全局卡表示意图

当一个指针引用到了 RSet 里面的一个区间时，包含该指针的堆内存块就会在 PRT 里面被标记。如果需要快速地扫描一张数据表，最好的方式是建立索引，一个粗粒度的 PRT 就是基于哈希表建立的。对于一个细粒度的 PRT 来说，哈希表内部的每一个入口对应一个区间，而区间内部的内存块索引也是存储在位图里面的。当细粒度 PRT 的最大值被突破的时候，我们就会开始采用粗粒度方式处理 PRT。

在垃圾回收过程中，当扫描 RSet 并且内存块确实存在于 PRT 里时，G1 GC 会在全局堆内存块数据表里标记对应的入口，这种做法避免了重新扫描这个内存块。G1 GC 会在回收循环阶段默认清除内存堆表，GC 输出日志如清单 4-2 所示，在 GC 线程的并行工作（主要包括根外部扫描、更新和扫描 RSet、对象拷贝、终止协议等）完成之后紧跟着的就是清除堆内存表标记（Clear CT）阶段。Update RS 和 Scan RS 对应的是 RSet 的更新和扫描动作。

代码清单 4-2　RSet 和 Clear CT 日志示例

```
12.540:[GC pause (G1 Evacuation Pause) (young), 0.0010622 secs]
[Parallel Time:0.5 ms, GC Workers:8]
```

```
   [GC Worker Start(ms):Min:0.2,Avg:0.3,Max:0.3,Diff:0.1,Sum:2.1]
   [Update RS(ms):Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.2]
   [Processed Buffers:Min:1,Avg:1.1,Max:2,Diff:1,Sum:9]
   [Scan RS(ms):Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.1]
   [Code Root Scanning(ms):Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.0]
   [Object Copy(ms):Min:0.0,Avg:0.0,Max:0.1,Diff:0.0,Sum:0.4]
   [Termination(ms):Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.0]
   [GC Worker Other(ms):Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.1]
   [GC Worker Total(ms):Min:0.3,Avg:0.4,Max:0.4,Diff:0.1,Sum:2.9]
   [GC Worker End(ms):Min:12664.5,Avg: 12664.5,Max: 12664.5,Diff:0.0]
   [Code Root Fixup:0.0ms]
   [Code Root Purge:0.0ms]
   [Clear CT:0.1ms]
   [Other:0.4ms]
      [Choose CSet:0.0ms]
      [Ref Proc:0.2ms]
      [Ref Enq:0.0ms]
      [Redirty Cards:0.1ms]
      [Humongous Reclaim:0.0ms]
      [Free CSet:0.1ms]
   [Eden:83.1M(83.1M)->0.0B(83.1M) Survivors:1024.1K->1024.1K
Heap:104.1M(104.1M)->21.0M(140.1M)
   [Times:user=0.00 sys=0.00,real=0.01 secs]
```

RSet 的作用是很明显的，但是在使用过程中我们也遇到了写保护和并行更新线程的维护成本。

OpenJDK HotSpot 的并行老年代和 CMS GC 都在执行 JVM 的一个对象引用写操作时使用了写保护机制，如代码 object.field=some_other_object。还记得我们对于每个区间是采用针对最小单元堆内存块进行管理的吗？这个写保护机制也会通过更新一个类似于堆内存块表的数据结构来跟踪跨年代引用。堆内存表在最小垃圾回收时会被扫描。写保护算法基于 Urs Holzle 的快速写保护算法，这个算法减少了编译代码时的外部指令消耗。

当跨越区间的更新发生的时候，G1 GC 会将这些对应的堆内存块放入一个缓存，我们可以称这个缓存为“更新日志缓存”，写入该缓存的方式和写入队列的方式一样。G1 GC 会使用一个专门的线程组去维持 RSet 信息，它们的职责是扫描“更新日志缓存”，然后更新 RSet。JDK8u45 采用选项-XX:G1ConcRefinementThreads 设置这个线程组的数量，如果你没有设置，那么默认采用-XX:ParallelGCThreads 选项。

一旦“更新日志缓存”达到了最大可用，它会被放入全局化的满载队列并启用一个新的缓存块。一旦更新线程在全局满载队列里面发现了入口，它们就开始并行处理整个满载缓存队列。

G1 GC 针对并行更新线程采用的是分层方法，为了保证更新速度会加入更多的线程，如果实在跟不上速度，Java 应用程序线程[1]也会加入战斗，但尽量不要出现这样的情况，这种情况是发生了线程窃取，会造成应用程序花费了本可以用于自身程序算法运行的能力。

4.2.7 并行标记循环[2]

从前面的介绍我们知道，G1 GC 的区间化设计需要一个完整的并行标记算法给予支撑。Taiichi Yuasa 的“Snapshot-At-The-Beginning（SATB）”标记算法解决了递增型标记和清除 GC 所需要的算法。SATB 算法聚焦于标记清洗 GC 的并行标记阶段，适用于 G1 GC 的区间化堆设计理念，并且解决了所有 HotSpot JVM 的 CMS GC 算法存在的重标记暂停延迟时间较长的缺陷。

G1 GC 设定了一个阈值，默认值是堆内存的 45%。这个阈值可以通过选项 IHOP 设定，IHOP 即-XX:InitiatingHeapOccupanyPercent。当超过阈值时，GC 会自动进入并行标记循环。并行标记循环的最终目标是在整个 Java 堆达到满载之前完成标记动作，同时为了确保 Java 应用线程可以并行运行，并行标记循环的内部标记任务被划分成了很多块任务，这样可以做到并行执行。这是一种常见的设计模式，在 GC 内部设计中我们已经看到了几次将较大的实体细分为更小实体的案例。从整个软件设计来看，诸如 HDFS 将 Block 设定为最小存储单元，也是基于这样的加速并行、最小单元互不干扰原理设计的。

SATB 算法通过创建一个对象图的方式完成堆内存逻辑上的快照，即将堆内存里所有需要回收的对象一一呈现在它的这张对象图上，这个标记的过程是在并行标记阶段完成的。该算法确保只有当前已经存在的对象才会被包含在对象图里面，并且在并行标记阶段会被进一步标记和跟踪状态，对于 Java 应用程序新创建的存活对象，在这一次并行标记阶段不会被计算在回收步骤里面。

SATB 算法维护的标记内容在数据结构上也分为两块，一块是已经完成的标记，另一块是即将进行的标记。这样的设计思想可以理解为复制算法，即前一块是已经完成标记，准备进入回收的区域，那么随着时间的推移，里面的对象将逐渐被回收，一旦全部回收完毕，就可以剔除前一块的指针，将后一块链接到前一块，然后再创建一个新的数据结构，用于存放下一块内容。对于这样的设计实现，G1 GC 堆区间有两个 TAMS 字段（Top-At-Mark-Start），分别被叫作前一个 TAMS（PTAMS）和后一个 TAMS（NTAMS）。

---

[1] 即 Mutator Threads。

[2] 即 Concurrent Marking Cycle。

如图 4-7 所示，在标记循环的开始阶段，NTAMS 字段被设置在每一个区间的顶端，依然占据空间的所有对象都有一个 TAMS 值，被放在 Top NTAMS 之上，其他在 TAMS 以下的对象都会被标记。

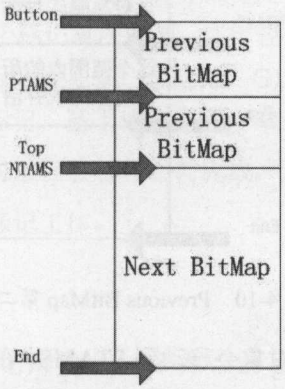


图 4-7　标记循环的开始阶段示意图

在图 4-8 里面，可以看出经过并行标记阶段的操作，一个堆区间有所变化，PTAMS 和 Bottom 之间的区域所包含的所有对象都会被标记，结果如图 4-9 所示。

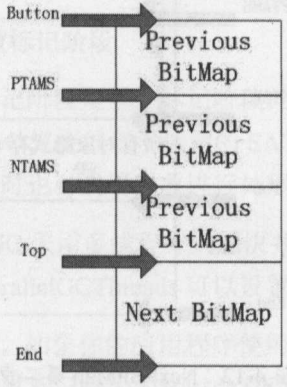


图 4-8　标记循环的中间阶段示意图

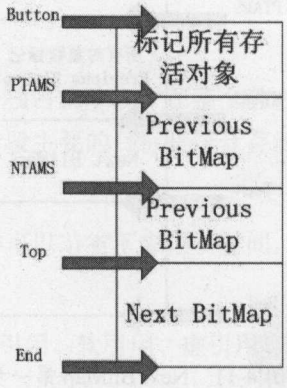


图 4-9　Previous BitMap 第一步

所有在 PTAMS 和 Top 之间的堆区间对象都会被标记为存活对象，如图 4-10 所示。

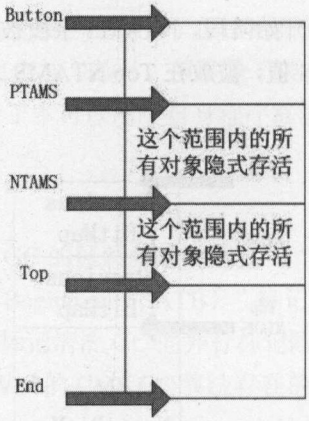


图 4-10　Previous BitMap 第二步

重标记暂停阶段，所有的存活对象会移动到 PTAMS，在 NTAMS 以下的区域就会产生新的一块内存区块，继续进行标记，如图 4-11 和图 4-12 所示。这一组循环也就是前面所说的两个区块之间的循环替代。

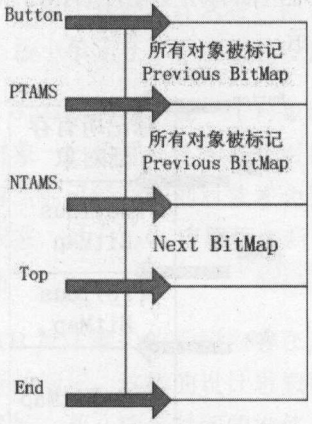


图 4-11　Next BitMap 第一步

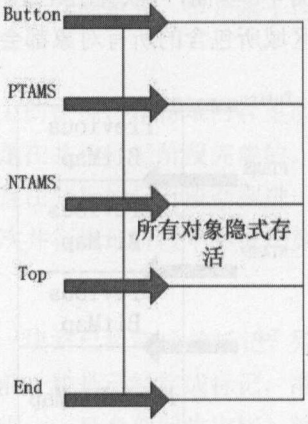


图 4-12　Next BitMap 第二步

并行标记循环的过程是初始标记阶段→根区间扫描阶段→并行标记阶段→重标记阶段→清除阶段，其中一部分是可以与应用程序并行执行的，一部分是独占式的。

1. 初始标记阶段

这个阶段是独占式的，它会停止所有的 Java 线程，然后开始标记根节点可及的所有对象。这个阶段可以和年轻代回收同时执行，这样的设计方式主要是为了加快独占阶段的执行速度。

在这个阶段，每一个区间的 NATMS 值会被设置在区间的顶部。

2. 根区间扫描阶段

设置了每个区间的 TAMS 值之后，Java 应用程序线程重新开始执行，根区间扫描阶段也会和 Java 应用程序线程并行执行。基于标记算法原理，在年轻代回收的初始标记阶段拷贝到幸存者区间的对象需要被扫描并被当作标记根元素，相应地，G1 GC 因此开始扫描幸存者区间。任何从幸存者区间过来的引用都会被标记，基于这个原理，幸存者区间也被称为根区间。

根区间扫描阶段必须在下一个垃圾回收暂停之前完成，这是因为所有从幸存者区间来的引用需要在整个堆区间扫描之前完成标记工作。

3. 并行标记阶段

首先可以明确的是，并行标记阶段是一个并行的且多线程的阶段，可以通过选项 -XX:ConcGCThreads 来设置并行线程的数量。默认情况下，G1 GC 设置并行标记阶段线程数量为选项-XX:ParallelGCThreads（并行 GC 线程[1]）的 1/4。并行标记线程一次只扫描一个区间，扫描完毕后会通过标记位方式标记该区间已经扫描完毕。

为了满足 SATB 并行标记算法的要求，G1 GC 采用一个写前 barrier 执行相应的动作。

4. 重标记阶段

重标记阶段是整个标记阶段的最后一环。这个阶段是一个独占式阶段，在整个独占式过程中，G1 GC 完全处理了遗留的 SATB 日志缓存、更新。这个阶段主要的目标是统计存活对象的数量，同时也对引用对象进行处理。

G1 GC 采用多线程方式加快并行处理日志缓存文件，这样可以节省下来很多时间，通过选项-XX:ParallelGCThreads 可以设置 GC 数量。

注意，如果你的应用程序使用了大量的引用对象，例如弱引用、软引用、虚引用、强引用，那么这个重标记阶段的耗时会有所增加。

5. 清除阶段

前面各个阶段在做的主要事情就是为了标记对象，那么为什么需要针对每一个区间进行标记呢？这是因为如果我们知道了每个区间的存活对象数量，如果这个区间没有一个存活对象，

---

[1] 并行 GC 线程在 JVM 开始阶段就已经开始执行。

那么就可以很快地清除 RSet，并且立即放入空闲区间队列，而不是将这个区间放入排队序列，等待一个混合垃圾回收暂停阶段的回收。RSet 也可以被用来帮助检测过期引用，例如，如果标记阶段发现所有在特定堆块上的对象都已经死亡，那么 RSet 可以快速清除这块堆块。

一句话总结，清除阶段会识别并清理完全空闲的区域。它是并发的清理，不会引起停顿。

4.2.8 评估失败[1]和完全回收

如果在年轻代区间或者老年代区间执行拷贝存活对象操作的时候，找不到一个空闲的区间，那么这个时候就可以在 GC 日志里看到诸如“to-space exhausted”这样的错误日志打印。来看一段日志，如清单 4-3 所示。

代码清单 4-3 to-space 异常日志

```
111.912:[GC pause(G1 Evacuation Pause)(young)(to-space exhausted),0.6772123 secs]
<snip>
[Evacuation Failure:330.1ms]
```

发生这个错误的同时，G1 GC 会尝试去扩展可用的 Java 堆内存大小。如果扩展失败，G1 GC 会触发它的失败保护机制并且启动单线程的完全回收动作。

在这个完全回收阶段，单线程会针对整个堆内存里的所有区间进行标记、清除、压缩等动作。在完成回收后，堆内存就完全由存活对象填充，并且所有的年龄代对应的区间都已经完成了压缩任务。

也正是因为这个完全回收是单线程执行的，所以当堆内存很大时势必耗时很长，所以需要谨慎使用，最好不要让它经常发生，以避免不必要的长时间的应用程序暂停。

4.3 G1 GC 使用场景

G1 的首要目标是为需要大量内存的系统提供一个保证 GC 低延迟的解决方案，也就是说堆内存在 6GB 及以上，稳定和可预测的暂停时间小于 0.5 秒。

如果应用程序具有如下的一个或多个特征，那么将垃圾收集器从 CMS 或 ParallelOldGC 切换到 G1 将会大大提升性能。

---

[1] 即 Evacuation Failure。

- Full GC 次数太频繁或者消耗时间太长。
- 对象分配的频率或代数提升（promotion）显著变化。
- 受够了太长的垃圾回收或内存整理时间（超过 0.5~1s）。

注意，如果正在使用 CMS 或 ParallelOldGC，而应用程序的垃圾收集停顿时间并不长，那么继续使用现在的垃圾收集器是个好主意。

4.4　G1 GC 论文原文翻译（部分）

4.4.1　开题

1. 摘要

本论文原文来自于 Sun 公司的 David Detlefs、Christine Food、Steve Heller、Tont Printezis。

垃圾优先（Garbage-First）是一款服务器类型的垃圾回收算法，服务目标是针对大内存环境且具有较多的处理器时，在达到高效的实时性目标的同时实现高吞吐能力。整个堆的操作，例如全局标记动作等都是和应用程序并行执行的，目的是避免堆或者存活对象大小成比例的中断。并行标记循环保证了回收的完整性，并且通过压缩评估方式找到回收的区间。这个评估过程是在多处理器情况下并行执行的，可以增大吞吐量。

2. 关键字

Concurrent Garbage Collection、Garbage Collection、Garbage-First Collection、Parallel Garbage Collection、Soft Real-Time Garbage Collection

3. 简介

Java 语言在服务端应用程序设计中被广泛应用。这些应用程序都具有大量存活的堆数据，并且都需要考虑多线程在多处理器上的并行特性。因此，吞吐量是至关重要的，但是它们通常又是受一定严格要求的实时性约束的，例如通信行业、电话处理应用等，即便是几秒钟的延时，对于客户来说感觉都是比较糟糕的。

Java 语言规范规定了一种叫作垃圾回收的机制来回收无用的堆内存。传统的“暂停式”回收器会影响应用程序的响应性，所以需要并行式或者递增式的回收器。在这样的回收器作用下，低暂停是以吞吐量作为交换代价的。因此，允许用户设定实时目标要求，保证回收器耗费时间

不超过多少毫秒。通过设定这样的目标，回收器可以尽量避免回收暂停，并且确保不是经常发生垃圾回收，按照这样的机制不会减少吞吐量或者增加不必要的垃圾占用空间。这篇论文介绍了一种叫作垃圾优先（Garbage-First）的回收算法（以下简称 G1 回收算法/回收器），这个算法尝试满足实时性要求的同时保持针对大的堆内存和高占有率的应用程序具有高吞吐量，同时可以运行在大型多处理器上面。

G1 回收器通过几个技术完成这些目标。堆内存被切分为等分大小的多个区间，这个技术很像火车一节节车厢的设计方式。然而，现有的对象空间回收器的记录集合是单向的，即是记录从老的区间到新的区间的单向指针，而不是双向的。G1 的记录集合记录了从所有区间来的指针，允许我们对堆区间进行选择，挑选我们认为合适的区间进行回收。特殊的应用写锁（mutator write barries）创建的日志记录需要通过一个并行线程处理，这样可以保持记录集合实时更新，帮助实现更短的回收操作，这是垃圾优先回收器的核心机制。

G1 使用了一个叫作开始快照（SATB）的并行标记算法。这个算法提供了全局对象可达的阶段性分析措施，从而保证完整性，即所有垃圾最终都会被证明是垃圾。并行标记线程会统计每个堆区间里面存活对象的数量，这个数量决定了哪一个区间会被回收，存活对象少、垃圾占有率高的区间容易被回收，这就是“垃圾优先”算法的原则。SATB 标记算法也会产生非常短的暂停时间。

G1 采用了一种梦幻般的机制去实现实时性目标。其他实时回收器通过标记回收中断拷贝独立对象的粒度，进而满足实时性的约束条件，这种方式存在时间和空间上的开销。相反地，G1 在堆区间里面执行粗粒度的对象拷贝。G1 回收器有一个比较精准的模型，这个模型控制了回收一个特定堆区间的代价，作为快速测量区间属性的一种方法。因此，回收器在暂停的限定时间范围内可以选择一系列的可回收区间（高概率）。而且，如果需要我们还可以对回收进行延迟，这样可以避免违反实施目标。我们相信抛弃强实时（Hard Real-Time）可以保证产生更好的吞吐量和空间使用率，这是许多应用程序的一个比较合适的折中方案。

4.4.2 数据结构/机制

在这章，会描述 G1 回收器使用的数据结构和算法机制。

1. 堆布局/堆区间/分配

G1 的堆内存被划分为相等大小的若干个区间（Regions），每一个区间是一个连续的虚拟

内存区块。一个堆区间的分配空间由递增的边界、顶部区域、分配和未分配空间等组成。一个区间是从当前分配好的内存里划分出来的存储空间，因此我们比较关心多处理器、应用线程分配、本地线程分配缓存（TLABs）、使用比较和交换算法、CAS 等等。然后它们在这些缓存内部交换分配对象、最小化分配区域的竞争。当我们选择的区间空间被全部填充满之后，一块新的区间空间会产生。空闲区间被管理在一个双向链表里面，这种做法让区间的分配时间保持在一个常数范围内。

大对象可能被直接存储在当前的区间里面，也可能在本地线程分配缓存以外。对象的大小超过堆区间 3/4 时，被定义为巨大的（Humongous）。

2. 记录集合维护

每一个区间对应拥有一个记录集合（Remembered Set），RSet 标识所有存储空间里存在指向存活对象的指针。维护这些记录集合时，需要引起数据变化的线程在它们做出指针修改时，即创建区间内部的指针时通知回收器。这种通知方式使用了 Card 表形式，每一个 512 个字节组成的 Card 存储在堆内存里，对应一个 1 字节的 Card 表入口。另外，存在一个全局的记录集合缓存。

记录集合本身是一系列 Card 的集合（通过 Hash Table 呈现）。事实上，由于并行算法的需要，每一个区间都有一个包括几个哈希表的数据对应着，一个并行 GC 线程会对记录集合进行更新操作，这个更新操作不会中途停止。记录集合的逻辑内容是每一个组件哈希表集合的整合。

记录集合写锁操作是在指针写入操作之后执行的。如果代码执行指针写操作，例如 x.f=y，然后注册 rX 和 rY，分别包含对象指针值 x 和 y，实现锁的伪代码如下所示：

```
rTmp = rX XOR rY
rTmp = rTmp>>LogOfHeapRegionsIZE
//Below is a conditional move instr
rTmp = (rY==NULL) then 0 else rTmp
if(rTmp==0) goto filtered
call rs_enqueue(rx)
filtered
```

如果一个写入操作在同一个堆区间里创建了一个对象指向另一个对象的指针，不需要在一个记录集合里做记录。

3. 评估暂停

在一个合适的时间点，我们会停止更新 RSet 的线程并启动评估暂停阶段。我们会选择一些

区间的集合信息，然后拷贝所有存活对象到堆内其他地方，以此评估这些区间，最后释放集合信息里面的区间。

假设一个多线程的应用程序运行在多处理器的机器上，如果使用一个单线程的垃圾回收器会导致应用程序出现性能瓶颈。因此需要尽全力去并行化一次评估暂停操作。

评估暂停的第一步是顺序排列 CSet 集合。接下来，评估暂停阶段最主要的并行阶段开始了。GC 线程通过竞争方式完成任务，这些任务包括扫描日志缓存文件用于更新日志集合、扫描记录集合和其他存活对象的根组、评估存活对象。在这些操作过程中，不存在同步方式以确保多个任务之间的数据共享保护，它们之间还是通过一个线程逐一执行的。

4. 年代型垃圾优先

年代型垃圾回收机制有几个优点。新分配的对象通常比老的对象更容易成为垃圾，并且新分配的对象更容易成为指针修改的目标。因此可以利用垃圾优先的特性，例如可以在区间被选择成为分配区间之后，试探性分配这个区间为年轻代区间。这个尝试会让这个区间成为下一个 CSet 的一员。作为回应，记录集合处理不再需要考虑在年轻代区间里执行修改。

注意，一个 CSet 可以包含年轻代和其他年代的区间。事实上垃圾优先可以有两种运行模式，年轻代优先和纯垃圾优先。年轻代模式是默认模式。

5. 并行标记

并行标记是整个系统的最重要组件，它不会受到 CSet 内区间的顺序影响，能够确保提供回收器完整信息。而且，它也会提供存活对象信息，用于垃圾优先回收。标记阶段确保从标记的开始阶段就定义垃圾对象，通过标记逻辑上的对象图快照方式保存存活对象关联信息。标记出来的对象都可以考虑作为存活对象。

4.4.3 未来展望

目前呈现给读者的 G1 垃圾回收器是针对并行和并发收集的组合收集器，针对大型多处理器及大内存的机器。G1 垃圾回收器的主要贡献包括以下内容。

- 这是一款高性能的以服务端方式设计的收集器，通过压缩方式有效地避免分配过程中针对细粒度队列的扫描耗时。这种方式作为回收器的简单实现，确保不出现潜在的碎片问题。
- 基于全局标记信息和其他容易获得的算法因子，根据 GC 性能优先回收区间，而不是简单地根据存活对象的总量进行判断，并且选择可以适配目标中断时间的集合进行回收。

- G1 试图预估未来的停顿时间，通过使用一个数据结构跟踪前一次暂停，可以避免出现未被实时性目标约束的出现，G1 允许在一个时间片上执行多次暂停。

开发团队对于 G1 未来的优化措施已经有几套方案。开发团队希望通过修改写入锁机制和记录集合的呈现方式提高记录集合的处理性能。开发团队未来还可能通过静态分析方式替换写入锁方式。最后，开发团队相信 G1 比其他的年代型 GC 具有更好的压缩时间逃逸分析。

4.5　本章小结

本章节对整个 G1 GC 相关概念进行了梳理及陈述，对于年轻代、大对象区间、混合回收、并行标记循环、Full GC 等都逐一做了完整的介绍，接下来会对如何利用 G1 GC 进行 Java 程序的性能优化进行一些讨论，也会在第 6 章对 JVM 诊断工具的使用方式进行图文并茂的介绍，最后对 G1 GC 的最初英文论文进行了原文翻译，介绍设计者的设计思维和发展过程。

5

第 5 章

G1 GC 性能优化方案

为什么互联网公司这么累？因为生产环境[1]出现的所有问题、缺陷，它们必须立即解决或者规避，否则出现的损失没有办法估算、挽回。性能优化工程师是所有工程师工种中最累的一个细分子工种，研发团队需要他们能够快速定位、解决软件产品运行过程中发现的问题，尤其是内存泄漏、应用程序无端挂起、堆内存溢出、空指针等情况，甚至于需要他们在缺少外部工具帮助调试的情况下，通过直接阅读代码找到问题原因并解决。我们会在这章对针对 G1 GC 的性能优化方案进行逐一深入的解释。

本章主要介绍和解决以下问题，这些也是全书的概念层面核心章节。

- 对 G1 GC 各种概念进一步深入学习。
- 深入学习 G1 GC 的设计理念。
- 深入运用 G1 GC 的特性。

---

[1] 即 Production Environment，是直接面向终端用户的软件产品、场景，有别于测试环境、开发环境。

5.1　G1 的年轻代回收

回顾前几章介绍的相关内容，分代式 GC 在年轻代的主要工作是回收 Eden 区和 Survivor 区。一旦 Eden 区被占满，年轻代 GC 就会启动。年轻代 GC 只处理 Eden 和 Survivor 区，回收后，所有的 Eden 区都应该被清空，但是应该至少仍然存在一个 Survivor 区（因为部分对象从 Eden 区复制到了 Survivor 区），类比其他的年轻代收集器，这一点似乎并没有太大的变化。另一个重要的变化是老年代的区域增多，因为部分 Survivor 区或者 Eden 区的对象可能会晋升老年代。

G1 年轻代包括串行和并行阶段。串行阶段是指任务需要一个接一个地被完成，只有在 STW 回收之后才能进行下一个任务；并行阶段指的是有多个 GC 工作线程启用[1]，当自己线程完成了分配的任务以后，可以从其他线程的工作队列里面窃取[2]任务执行。

第 3 章简单地介绍了 G1 GC 的输出日志，这里我们深入地介绍，注意，我们采用的是 JDK8，后续不再赘述。

假设我们现在有 GC 输出日志（采用-XX:+PrintGCDetails、-XX:+UseG1GC 选项后的输出），如代码清单 5-1 所示。

代码清单 5-1　-XX:+UseG1GC 运行输出

```
   100.010:[GC pause (G1 Evacuation Pause) (young),0.05213secs]
   [Parallel Time: 50.2ms, GC Workers:8]
      [GC Worker Start(ms): Min: 100010.3, Avg: 100010.3, Max: 100010.4,
Diff:0.1]
      [Ext Root Scanning(ms):Min:0.1,Avg:0.2,Max:0.2,Diff:0.1,Sum 1.2]
      [Update RS(ms): Min 12.8, Avg:13.0,Max:13.2,Diff:0.4,Sum 103.6]
        [Processed Buffers: Min:15,Avg 16.0,Max:17,Diff: 2,Sum: 128]
   [Scan RS(ms): Min:13.4,Avg:13.6,Max:13.7,Diff:0.3,Sum:109.0]
   [Code Root Scanning(ms) : Min:0.0,Avg:0.0,Max: 0.0,Diff:0.0,Sum:0.1]
   [Object Copy(ms): Min:25.1,Avg:25.2,Max:25.2,Diff:0.1,Sum:201.5]
   [Termination(ms): Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.1]
   [GC Worker Other(ms):Min:0.0,Avg:0.1,Max:0.1,Diff:0.1,Sum:0.4]
   [GC Worker Total(ms):Min:51.9,Avg:52.0,Max:52.1,Diff:0.1,Sum:416.0]
   [GC Worker End(ms):Min:108867.5,Avg: 108867.5,Max: 108867.6,Diff:0.1]
```

[1] 年轻代期间的线程数量可以通过选项-XX:ParallelGCThreads 配置。

[2] 通常在技术书里面用 Stealing from 这个词组表示窃取，很多计算机概念里面都有这个动词。

```
   [Code Root Fixup: 0.1ms]
   [Code Root Purge: 0.0ms]
   [Clear CT:0.2 ms]
   [Other: 2.0ms]
      [Choose CSet: 0.0ms]
      [Ref Proc: 0.1ms]
      [Ref Enq: 0.0ms]
      [Redirty Cards: 0.2ms]
      [Humongous Reclaim:0.0ms]
      [Free CSet: 1.2ms]
   [Eden: 501.0M(501.0M)->0.0B(502.0M) Survivors:23.0M->31.0M
Heap:841.2M(1024.0M)->310.4M(1024.0M)]
```

从第 1 行 100.010:[GC pause (G1 Evacuation Pause) (young),0.05213secs]我们可以看出，关键字是 G1 Evacuation Pause 和 Young，指的是年轻代收集停顿，这个停顿是从 100.010 这个时间戳开始的，一共执行了 0.05213s。

1. Start of All parallel Activities

清单 5-1 所示的第 2 行显示了花费在并行阶段上的运行总时间，以及 GC 工作线程数量：

```
   [Parallel Time: 50.2ms, GC Workers:8]
```

这一行显示这次程序在并行阶段一共花费了 50.2ms，GC 最大并行工作线程是 8 个，也就是说在 8 个工作线程并行运行的情况下，这一次年轻代 GC 回收工作一共耗时 50.2ms。

以下如代码清单 5-2 所示为 8 个工作线程执行的主要并行工作。

代码清单 5-2　8 个工作线程执行的主要并行工作

```
   [GC Worker Start(ms): Min: 100010.3, Avg: 100010.3, Max: 100010.4,Diff:0.1]
   [Ext Root Scanning(ms):Min:0.1,Avg:0.2,Max:0.2,Diff:0.1,Sum 1.2]
   [Update RS(ms): Min 12.8, Avg:13.0,Max:13.2,Diff:0.4,Sum 103.6]
   [Processed Buffers: Min:15,Avg 16.0,Max:17,Diff: 2,Sum: 128]
   [Scan RS(ms): Min:13.4,Avg:13.6,Max:13.7,Diff:0.3,Sum:109.0]
   [Code Root Scanning(ms) : Min:0.0,Avg:0.0,Max: 0.0,Diff:0.0,Sum:0.1]
   [Object Copy(ms): Min:25.1,Avg:25.2,Max:25.2,Diff:0.1,Sum:201.5]
   [Termination(ms): Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.1]
   [GC Worker Other(ms):Min:0.0,Avg:0.1,Max:0.1,Diff:0.1,Sum:0.4]
   [GC Worker Total(ms):Min:51.9,Avg:52.0,Max:52.1,Diff:0.1,Sum:416.0]
   [GC Worker End(ms):Min:108867.5,Avg: 108867.5,Max: 108867.6,Diff:0.1]
```

G1 GC 的每一次 GC 工作都是由 GC Worker Start 和 GC Worker End 这两个标签分别表示并

行阶段的开始和结束，也就是说具体内容都是在其内部的。GC Worker Start 这一行的 Min 时间戳表示第一个工作线程开始的时间（一共有 8 个工作线程，每一个开始的时间总有差别，这里选最早开始的那个）。相应地，Max 指的是最后一个工作线程完成分配给它的所有任务的时间戳[1]。这一行除了 Min 和 Max 以外，还有 Diff 和 Avg 两个标签。Diff 指的是 Max 和 Min 之间的差距，Avg 指的是平均值。在 Min 或者 Max 之间存在较大的差值，也许是由于窃取线程存在的异常造成的，这时候需要进一步分析并行阶段的并行工作是否存在问题。

2. External Root Regions

接下来了解一下外部根区间扫描[2]对应的 GC 输出，在前面如代码清单 5-2 所示的输出中，具体看下面这一行：

```
[Ext Root Scanning(ms):Min:0.1,Avg:0.2,Max:0.2,Diff:0.1,Sum 1.2]
```

这一行里出现的 Min 和 Max 和前面解释的一样，也是 8 个并行线程最早开始和最迟结束时间。

我们主要观察是否 Diff 远大于 0 或者 Min、Max、Avg 这三个变量是较大的值的情况。

外部根区间扫描指的是从根部开始扫描通过 JNI 中本地的类中调用 Malloc 函数分配出的内存。这个步骤是并行任务的第一个任务。这个阶段堆外（off-heap）根节点被开始扫描，这些扫描范围包括 JVM 系统字典[3]、VM 数据结构、JNI 线程句柄、硬件注册器[4]、全局变量，以及线程栈根部[5]等，这个过程主要是为了找到并行暂停阶段是否存在指向当前收集集合（CSet）的指针。

这里还是要提醒大家注意 Diff 这个值，如果这个值很大，说明工作线程启动和结束之间的耗时较长，也就可以推测出来也许工作任务分配不均匀。注意，这只是猜测，只能通过 GC 日志输出来进一步地判断，也可以通过阅读 Java 应用程序源代码来查找问题，或许你的代码需要重构都未可知。

这里还有一个情况需要引起大家的重视，就是查看工作线程是否在处理一个单一的根节点时耗时过长，导致感觉类似挂起的现象。这个现象可以通过查看工作线程对应的“termination”

1　也就是说，这是一个并行工作线程的执行最长路径。

2　即 External Root Region Scanning。

3　即 System Dictionary。

4　即 Hardware Registers。

5　即 Thread Stack Roots。

日志看出来。如果存在这个现象，你需要去查看是否存在比较大的系统字典（JVM System Dictionary），如果这个系统字典被当成了一个单一根节点进行处理，那么当存在大量的加载类时就会出现较长时间的耗时。

3. Rememebered Sets and Processed Buffers

接下来再来看一下 RSet 对应的 GC 输出，如代码清单 5-2 包含的内容，具体看下面这一行：

```
[Update RS(ms): Min 12.8, Avg:13.0,Max:13.2,Diff:0.4,Sum 103.6]
[Processed Buffers: Min:15,Avg 16.0,Max:17,Diff: 2,Sum: 128]
```

第 4 章我们详细介绍过 RSet，RSet 帮助维护和跟踪指向 G1 区间的引用，而这些区间本身拥有这些 RSet。还记得我们在第 4 章介绍过的并行 Refinement 线程吗？这些线程的任务是扫描更新日志缓存，并且更新区间的 RSet。为了更加有效地支援这些 Refinement 线程的工作，在并行回收阶段，所有未被处理的缓存（已经有日志写在里面了）都会被工作线程拿来处理，这些缓存也被称为日志里面的处理缓存[1]。

为了限制花费在更新 RSet 上的时间，G1 通过选项-XX:MaxGCPauseMills 设置了目标暂停时间，采用相对于整个停顿目标时间百分比的方式，限制了更新 RSet 花费的总时长，让评估暂停阶段把最大量的时候花费在拷贝存活对象上。这个目标时间默认为整个停顿时间的 10%[2]，例如整个停顿时间是 10s，那么花费在更新 RSet 上的时间最大为 1s。G1 GC 的设计目标是让更多的停顿时间花费在拷贝存活对象上面，因此暂停时间的 10%被用于更新 RSet 也是比较合理的，百分比大了，花在干具体业务（各阶段拷贝存活对象）上的时间也就少了。

如果你发现这个值不太准确或者不符合你的实际需求，这里可以通过更新选项-XX:G1RSetUpdatingPauseTimePercent 来改变这个更新 RSet 的目标时间值。切记，如果你改变了花费在更新 RSet 上的时间，那你必须有把握工作线程可以在回收暂停阶段完成它们的工作，如果不能，那这部分工作[3]会被放到并行 Refinement 线程里面去执行，这会导致并行工作量增加、并行回收次数增多。最坏的情况是如果并行 Refinement 线程[4]也不能完成任务，那么 Java 应用程序就会被暂停，原本负责执行 Java 应用程序的资源就会直接接手任务，这个画面“太美”，不敢看！大家要尽量避免这种情况发生。

---

1 即 Processed Buffers。

2 停顿阶段大多数时间应该被花费在拷贝存活对象上，所以 10%留给更新 RSet 也差不多了。

3 即 Log Buffer Update 工作。

4 可以通过选项-XX:G1ConcRefinementThreads 设置线程数量，默认和-XX:ParallelGCThreads 相同，并且两个值互相关联。

注意，-XX:G1ConcRefinementThreads 选项的值默认和-XX:ParallelGCThreads 的值一样，这意味着对于-XX:ParallelGCThreads 选项的修改会同样改变-XX:G1ConcRefinementThreads 选项的值。

在当前 CSet 里面回收之前，CSet 内部的每个区间的 RSet 都需要被扫描，主要目的是找到 CSet 区间内部的引用关系。一个有较多存活对象的区间容易导致 RSet 的粒度变细，即每个区间对应的表格会从粗粒度变为细粒度，也可以理解为里面对象增多后扫描一个 RSet 需要更长的扫描时间，这样你就会看到更多的时间被花费在了扫描 RSet 上面。也可以理解为扫描时间取决于 RSet 数据结构的粗细粒度。

这种场景下，你会看到“Scan RS”这一栏耗时拉长了，因为扫描时间依赖于 RSet 数据结构的粗细粒度。来看一下代码清单 5-2 里面扫描 RS 的时间：

```
[Scan RS(ms): Min:13.4,Avg:13.6,Max:13.7,Diff:0.3,Sum:109.0]
```

这里的扫描针对的是从当前 CSet 里收集区间（Region）之前，这些区间对应的 RSet 必须被全部扫描。这个扫描的过程实际上是一个针对 PRT（Per-Region Table）的精炼过程，即从粗粒度到细粒度的过程。那么就容易理解了，如果是粗粒度，这个过程就需要花费较多的时间，这个实例是花费了 0.3ms。

另一个和 RSet 相关的并行任务是代码根扫描，在这个阶段主要是为了查找指向当前 CSet 的引用对象，如代码清单 5-2 里的 GC 日志：

```
[Code Root Scanning(ms) : Min:0.0,Avg:0.0,Max: 0.0,Diff:0.0543861,Sum:0.1]
```

在 HotSpot 的早期版本中，代码根扫描是一个单线程、不分段操作，就是说一个线程从头干到尾，其他工作线程的中断时间很长。一个全量代码根扫描会暂停工作线程的执行，导致整体暂停时间拉长。当代码根扫描可以被切分为多段并且并行执行之后，只有在特定情况下才会进行全代码扫描，否则一般都是仅仅从编译代码开始扫描 RSet 的引用，而不是扫描 Nmethods。

这里提到了 Nmethods 这个概念，它可以让开发人员对 Java 方法进行动态编译，不要与本地方法（Native Methods）混淆，本地方法指的是一个 JNI 方法。Nmethods 包含了辅助信息，例如除了生成代码以外，还包括常量池等辅助信息。为了减少 Nmethods 的扫描时间，只有 CSet 内部的 RSet 才会被编译器扫描对象引用情况，而不是由 Java 应用程序线程扫描所有的对象引用信息。

4. Summarizing Remembered Sets

-XX:+G1SummarizeRSetStats 选项用于统计 RSet 的密度数量（细粒度或者粗粒度），这个

密度帮助决定是否并行 Refinement 线程有能力去应对更新缓存的工作，并且收集更多关于 Nmethods 的信息。这个选项每隔 *n* 次 GC 暂停收集一次 RSet 的统计信息，这个 *n* 次由选项 -XX:G1SummarizeRSetStatsPeriod=n 决定，也是需要通过选项进行设置的。

注意，-XX:+G1SummarizeRSetStats 选项是一个诊断选项，因此必须启用-XX:+UnlockDiagnosticVMOptions 选项才可以启用-XX:+G1SummarizeRSetStats 选项。

如代码清单 5-3 所示，这是一次 GC 暂停（达到了 *n* 次要求）时候输出的 RSet 统计信息。

代码清单 5-3　GC 暂停阶段输出的 RSet 统计信息日志

```
Before GC RS summary
Recent concurrent refinement statistics
Processed 23270 cards
Of 96 completed buffers:
          96 (100.0%) by concurrent RS threads
           0 ( 0.0%) by mutator threads
     Did 0 coarsenings.
     Concurrent RS threads times (s)
       1.11    1.11   1.11   1.11   1.11   1.11    1.11    0.00
     Concurrent sampling threads times (s)
            0.96
    Current rem set statistics
      Total per region rem sets sizes = 4380K. Max=72K.
         767K (17.5%)  by 212 Young regions
         29K (0.7%) by 9 Humongous regions
         2151K (49.1%) by 648 Free regions
         1431K (32.7%) by 155 Old regions
      Static structures = 256K,free_lists=OK.
        816921 occupied cards represented.
          13670 (1.7%)  entires by 212 Young regions
             4 (0.0%) entires by 9 Humongous regions
             0 (0.0%) entires by 648 Free regions
          803212 (98.3%) entires by 155 Old regions
      Region with largest rem set = 4:(0)[0x00000006c0400000,
0x00000006c0500000,0x00000006c0500000],
     Size = 72K,occupied=190K.
     Total heap region code root sets sizes = 40K.  Max = 22K.
            3K (8.7%) by 212 Young regions
            0K (0.3%) by 9 Humongous regions
            10K (24.8%) by 648 Free regions
```

```
                27K (66.2%) by 155 Old regions
         1035 code roots represented.
                5 (0.5%) elements by 212 Young regions
                0 (0.0%) elements by 9 Humongous regions
                0 (0.0%) elements by 648 Free regions
                1030 (99.5%) elements by 155 Old regions
   Region with largest rem set = 4:(0)[0x00000006c0400000,0x00000006c0500000,
0x00000006c0500000],
      Size = 22K,num_elems=0.

   After GC RS summary
   Recent concurrent refinement statistics
   Processed 3782 cards
   Of 26 completed buffers:
           26 (100.0%) by concurrent RS threads
            0 ( 0.0%) by mutator threads
       Did 0 coarsenings.
       Concurrent RS threads times (s)
         0.00     0.00   0.00   0.00   0.00   0.00    0.00    0.00
       Concurrent sampling threads times (s)
              0.00
      Current rem set statistics
        Total per region rem sets sizes = 4329K. Max=73K.
           33K (0.8%)  by 10 Young regions
           29K (0.7%) by 9 Humongous regions
           2689K (62.1%) by 810 Free regions
           1577K (36.4%) by 195 Old regions
        Static structures = 256K,free_lists=63K.
          805071 occupied cards represented.
                0 (0.0%)  entires by 10 Young regions
                4 (0.0%) entires by 9 Humongous regions
                0 (0.0%) entires by 810 Free regions
             805067 (100.0%) entires by 195 Old regions
        Region with largest rem set = 4:(0)[0x00000006c0400000,
0x00000006c0500000,0x00000006c0500000],
      Size = 73K,occupied=190K.
      Total heap region code root sets sizes = 40K.  Max = 22K.
                0K (0.8%) by 10 Young regions
                0K (0.3%) by 9 Humongous regions
                12K (30.9%) by 810 Free regions
                27K (68.0%) by 195 Old regions
```

```
        1036 code roots represented.
               2 (0.2%) elements by 10 Young regions
                0 (0.0%) elements by 9 Humongous regions
                0 (0.0%) elements by 810 Free regions
                1034 (99.8%) elements by 195 Old regions
   Region with largest rem set = 4:(0)[0x00000006c0400000,0x00000006c0500000,
0x00000006c0500000],
      Size = 22K,num_elems=0.
```

我们需要重点关注的输出如代码清单 5-4 所示。

代码清单 5–4　重点关注

```
   Processed 23270 cards
   Of 96 completed buffers:
            96 (100.0%) by concurrent RS threads
             0 ( 0.0%) by mutator threads
       Did 0 coarsenings.
```

这个输出主要是 GC 暂停前后的输出描述。Processed cards 标签总结了并行 Refinement 线程完成的工作，当然也会包含 Java 应用程序线程完成的工作，在上面的示例中，96 个完成的缓存包含了 23270 个处理信息（cards），并且 100%都是由并行 Refinement 线程完成的。Did 0 coarsenings 表示没有 RSet 粒度处理。

清单 5-3 日志里面的其余部分描述了并行 RSet 时间以及当前 RSet 的统计信息，包含大小、每个不同类型（年轻代、空闲、大对象、老年代等）的区间占用的表格数量（这里把数据信息想象成一个个表格，对应不同的内容）。这些日志可以帮助你去弄清楚 RSet 引用的每一个区间类型，同时也让你清楚每个区间类型的引用数量。注意，对于四个潜在提升区域（RSet 粒度、更新 RSet、扫描 RSet、扫描 Nmethods 引用）的研究有助于理解应用程序的优化方式。

5. Evacuation and Reclamation

那么既然 G1 知道并行回收暂停阶段的 CSet 的详细信息，伴随着 CSet 内部引用集合的收集完成，我们进入了暂停阶段最为耗时的步骤，即评估 CSet 区间内部存活对象的详细情况并准备回收使之成为空闲空间。一般来说，拷贝存活对象所花费的时间占了整个暂停阶段的大头。存活对象的拷贝动作首先需要找到目标区间，然后将需要评估的存活对象拷贝到这些区间内部的本地线程 GC 分配缓存（GCLABs[1]）里。

[1] 即 thread-local GC allocation buffers。

工作线程通过互相竞争的方式实现对于拷贝对象的线性表达，即始终保持一个向前的指针，老的对象镜像指向新分配的拷贝区间指针，这有点像铺地砖，一块接着一块铺，不会随便在中间插入一块，否则工作就不好评估了。在工作线程窃取逻辑的帮助下，只有一个线程最终完成了针对一个对象的扫描和复制工作，这样可以帮助在多个工作线程之间完成负载均衡。拷贝对象的耗时如下所示：

```
[Object Copy(ms): Min:25.1,Avg:25.2,Max:25.2,Diff:0.1,Sum:201.5]
```

注意，G1 GC 使用这个对象的拷贝时间作为预测拷贝一个单一区间时间的权重值。用户如果发现预计时间跟不上目标时间（Target Time），那么可以通过修改年轻代大小的方式来减少区间大小，以便更快速地扫描、复制存活对象，实现预测时间在目标暂停时间范围内的目的。

6. Termination

接下来进行的是一个叫作中断的操作，这个操作实际上是一个过程，包含了检查自己线程内部工作队列是否空了，如果空了，就准备到其他线程去窃取点任务做，这样可以加快执行速度。如果什么工作都窃取不到，那么就调用中断操作，终结自己。“Termination（中断）”这个标签标记了每个工作线程花费在终止协议上的时间。一个 GC 的工作线程投身于单个根扫描之后，它会拖延队列内部所有任务的完成时间，并且最终导致中断时间晚于预期。如果所有的工作线程都存在这个问题，那么会导致中断时间（Termination 标签）较晚。

代码清单 5-2 里面对应的一行日志：

```
[Termination(ms): Min:0.0,Avg:0.0,Max:0.0,Diff:0.0,Sum:0.1]
```

这一行就是对应的中断花费时间。

7. Parallel Activity Outside of GC

中断标志着评估/回收暂停阶段内的并行工作线程工作的结束。在中断日志之后，我们看到了代码清单 5-2 里面有一行以 GC Worker Other 开头的日志：

```
[GC Worker Other(ms):Min:0.0,Avg:0.1,Max:0.1,Diff:0.1,Sum:0.4]
```

注意，这个阶段花费的时间不是针对 GC 的，所以叫作“Other（其他）”。正如字面意思，这个过程虽然还是属于 GC 暂停阶段，但实际上已经把时间交给了 GC 以外的工作，比如 JVM 编译。所以前面的各个步骤属于 GC 常规工作，这个过程属于非常规工作花费的时间。如果我们发现这个阶段耗时很长，就应该去检查 GC 之外的工作，比如可能引起耗时长的是编译过程中采用了错误的编译选项。

8. Summarizing All Parallel Activities

GC 工作线程结束标签之前还剩下最后一个统计总花费时间的工作步骤，以“GC Worker Total”标记，包含了寻常和非寻常的 GC 工作线程的时间总和，如清单 5-2 里节选的这一行：

```
[GC Worker Total(ms):Min:51.9,Avg:52.0,Max:52.1,Diff:0.1,Sum:416.0]
```

9. Start of All Serial Activites

并行阶段随着关键字“GC Worker End”的出现也跟着结束了，接下来是以关键字“Code Root Fixup,Code Root Purge,Clear CT”开始的串行阶段。在这个阶段，GC 主线程[1]更新代码根源（Code Roots）到新的标记对象地址，并且清除代码根源设置表。“Clear CT”阶段是一个并行阶段（工作线程可以并行执行），它会清除所有扫描 RSet 时特意遗留的标签，这个标签是用来告诉扫描线程某一个区间已经完成了扫描，避免重复劳动，这个标签被保留在全局卡表（Global Card Table）中。

如代码清单 5-2 的节选：

```
[Code Root Fixup: 0.1ms]
[Code Root Purge: 0.0ms]
[Clear CT:0.2 ms]
```

在一个混合回收暂停阶段，虚拟机线程更新对象位置的过程（即 Code Root Fixup）会包含更新没有评估区间所花费的时间。

10. Other Serial Activities

串行阶段的最后一个步骤以“Other”标签开头。这个过程完成的工作包括：选择回收阶段的 CSet[2]、引用处理和入队列、Card Redirtying、声明可用的空闲大对象区间、回收阶段后释放 CSet 等。花费的时间主要如代码清单 5-2 的节选：

```
   [Other: 2.0ms] [Choose CSet: 0.0ms] [Ref Proc: 0.1ms]  [Ref Enq: 0.0ms]
[Redirty Cards: 0.2ms]  [Humongous Reclaim:0.0ms] [Free CSet: 1.2ms]
```

本例一共花费了 2ms。

注意，针对年轻代回收而言，所有的年轻代区间都会被回收，所以也就没有了“选择”这样的规则设置。选择是相对于混合回收阶段的，作为混合回收的一个重要组成因子，减少混合

---

[1] 即 VM 线程，用于在保护模式（SafePoint）下执行 GC 的 VM 操作。

[2] 这个过程的花费时间只会在混合 GC 阶段存在，年轻代回收阶段已经包含在前面的过程了，所以时间为 0。

回收的整体耗时。

Java 对象的引用入队和处理过程是针对软引用、弱引用、虚引用、final，以及 JNI 等引用对象的。具体关于这些对象的引用类型可以参阅第 1 章，也可以看我的《大话 Java 性能优化》这本书，内容相对较全面。引用入队的过程可能需要更新 RSet，即本例中输出的 Ref Proc:耗时 0.1ms，Ref Enq 耗时 0.0ms，分别代表引用操作耗时和引用入队耗时。因此，这个更新操作需要写入日志，并且它们对应的表格也需要被标记为“已写”。这个耗时就是我们列出的写入表格时间，即 0.2ms。大对象回收是从 JDK8u40 开始引入的，如果一个大对象从根集合或者年轻代区间开始就没有引用关系了，或者在 RSet 里面没有引用了，那么可以在评估阶段就回收这个大对象。

整体来看，这个“Other”过程应该是比较短暂的，因为它基本都在处理 JNI 句柄以及类似的工作。比如说你释放 CSet 耗时较长，那么说明 CSet 暂停阶段耗时较长；Ref Proc 和 Ref Enq 耗时较长则意味着应用程序包含了大量的引用；大对象回收耗时较长就意味着有很多短命的大对象，这个一般来说是不对的，因为大对象占用了很多资源，如果经常回收，它的设计理念就被曲解了。总之，所有的长时间都需要给出合理的解释。

5.2　年轻代优化[1]

G1 GC 存在很大的优化潜质。我们需要深入去了解每一个选项的默认值和调整值之间的关系，即知道如果调整了这个选项的值，会给 G1 GC 带来什么样的不同。比如几个常用的选项：-XX:MaxGCPauseMills，默认值是 200ms，代表目标停顿时间；-XX:G1NewSizePercent，默认值是 5%，代表初始化年轻代占用整个堆内存的百分比；-XX:G1MaxNewSizePercent，默认值是 60%，代表年轻代的上升空间，即最大可以占用堆内存的百分比。这三个选项会帮助提升或者下降年轻代，是基于初始值和浮动上限、目标停顿时间、前一次拷贝时间的加权平均值的。总的来说，这个还是一个经验活，需要不断地尝试才能找到最适合自己应用程序的选项设置规律。

我们使用选项-XX:MaxGCPauseMills，并且开启-XX:+PrintAdaptiveSizePolicy 选项，运行结果如代码清单 5-5 所示。

代码清单 5-5　-XX:MaxGCPauseMills 运行结果 1

```
6.311:[GC pause(G1 Evacuation Pause)(young) 6.311:[G1Ergonomics(CSet
```

[1] 即 Young Generation Tunables。

```
Construction) start choosing CSet,_pending_cards:5800,predicated base
time:20.39ms,remaining time:179.61ms,target pause time:200.00ms]
   6.311:[G1Ergonomics(CSet Construction) add young regions to CSet,eden:225
regions, survivor:68 regions, predicated young region time:202.05ms]
   6.311:[G1Ergonomics(CSet Construction) finish choosing CSet, eden:225
regions, survivors:68 regions, old: 0 regions, predicated pause time:222.44ms,
target pause time: 200.00ms],0.1126132 secs]
```

从上面的 GC 运行输出可以看到，最后一行显示目标停顿时间是 200ms，这一次年轻代的 GC 预计时间是 222.44ms，实际运行时间为 112.61ms。日志的第 2 行可以看到一共有 225 个 Eden 区间，可以通过减小目标时间的方式加快实际的执行时间（本例改为 50ms），同时也增加了更多的年轻代 Eden 区间，运行结果如代码清单 5-6 输出日志所示。

代码清单 5-6　-XX:MaxGCPauseMills 运行结果 2

```
   6.317:[GC pause(G1 Evacuation Pause)(young) 6.317:[G1Ergonomics(CSet
Construction) start choosing CSet,_pending_cards:9129,predicated base
time:14.46ms,remaining time:35.54ms,target pause time:50.00ms]
   6.317:[G1Ergonomics(CSet Construction) add young regions to CSet,eden:284
regions, survivor:16 regions, predicated young region time:60.90ms]
   6.317:[G1Ergonomics(CSet Construction) finish choosing CSet, eden:284
regions, survivors:16 regions, old: 0 regions, predicated pause time:75.36ms,
target pause time: 50.00ms],0.0218629 secs]
```

从上面的日志输出变化可以看到，只是调整了目标停顿时间，预测停顿时间减小为 75.36ms，对应的实际执行时间减小为 21.86ms，Eden 区间增多为 284 个，这样就加快了停顿阶段的回收速度。

如果这个时候调整年轻代的占用空间，即设置选项-XX:G1NewSizePercent 为 10%（默认值是 5%），来看一下代码清单 5-7 的输出。

代码清单 5-7　-XX:MaxGCPauseMills 运行结果 3

```
   6.318:[GC pause(G1 Evacuation Pause)(young) 6.318:[G1Ergonomics(CSet
Construction) start choosing CSet,_pending_cards:5518,predicated base
time:10.00ms,remaining time:40.00ms,target pause time:50.00ms]
   6.318:[G1Ergonomics(CSet Construction) add young regions to CSet,eden:475
regions, survivor:25 regions, predicated young region time:168.35ms]
   6.318:[G1Ergonomics(CSet Construction) finish choosing CSet, eden:475
regions, survivors:25 regions, old: 0 regions, predicated pause time:178.35ms,
target pause time: 50.00ms],0.0507471 secs]
```

我们可以看到，随着年轻代的增大，CSet 对应的年轻代区间也增加了很多（从 284 个增加到 475 个），对应的实际停顿时间也延长到 50.75ms（目标停顿时间是 50ms）。

5.3　并行标记阶段优化[1]

并行标记阶段与老年代、大对象区间一起计算，占用空间是整个 Java 堆区的对应百分比，由选项-XX:InitiatingHeapOccupancyPercent 进行设置（默认值是 45%），这个选项决定了什么时间初始化并行标记循环。注意，CMS GC 的标记循环初始化阶段只和老年代大小相关。

并行标记循环从初始化标记暂停阶段开始，而这个初始化标记暂停又和一个年轻代回收暂停同步开始。初始化标记暂停标志着回收循环的开始，紧随其后的是其他并列任务，例如根区间扫描、并行标记、存活对象统计、最终标记，以及清除阶段等。

并行标记阶段一般由下面的 GC 日志语句开始启动，比如这样的示例。

```
277.559:[GC pause (G1 Evacuation Pause) (young) (initial-mark),0.0960289 secs]。
```

我们假设年轻代回收暂停、混合回收暂停、并行标记循环暂停的时间如图 5-1 所示，其中并行标记循环又包含了初始化标记、重标记、清除等阶段。

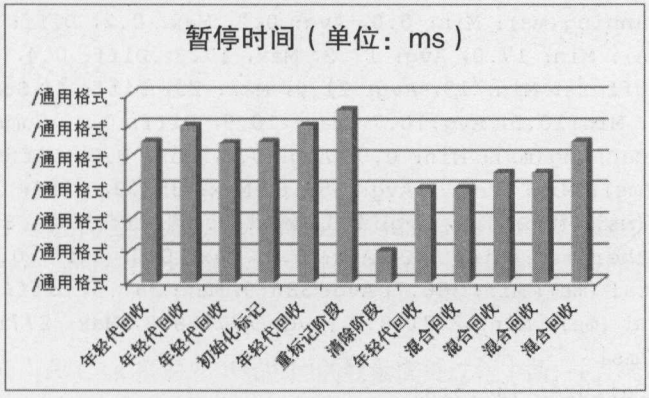


图 5-1　各阶段耗时图

并行标记任务和年轻代回收是同步进行的，所以它经常会被年轻代回收暂停打断，同时它

[1] 即 Concurrent Marking Phase Tunables。

的耗时也和应用程序的存活对象有关，即应用程序的存活对象图[1]越大，耗时就越长，反之亦然。

并行标记循环必须在一个混合回收开始前完成，这是混合回收循环运行的必要条件之一。但也不是说并行标记循环结束之后混合回收阶段就开始了，如图 5-1 所示，几个年轻代回收完成以后，就可以进入到一个并行标记循环。从初始化标记开始，到清除阶段，中间还夹杂着年轻代回收（这个年轻代回收主要用来评估触发了一个混合回收的阈值，即判断混合回收是否启动），所以证明年轻代回收和并行标记阶段可以并行执行。当并行标记阶段完成之后，根据对象存活的阈值判断是否需要进行混合回收，这时候混合回收阶段就启动并回收整个目标 CSet 区间的垃圾。

并行标记任务的耗时牵一发而动全身，它的耗时如果较长，整个循环就会相应地耗时较长，混合回收就会相应地延期开始，最终导致整个评估失败。如果你在 GC 日志里看到了“to-space exhausted”这样的文字，就说明回收失败了，整个回收耗时在“other”部分打印。如代码清单 5-8 所示。

代码清单 5-8　回收失败日志案例

```
276.731:[GC pause (G1 Evacuation Pause) (young) (to-space exhausted), 0.82729 secs]
 [Parallel Time: 387.0ms, GC Workers:8]
 [GC Worker Start(ms): Min: 276731.9, Avg: 276731.9, Max: 276732.1, Diff: 0.2]
 [Ext Root Scanning(ms): Min: 0.0, Avg: 0.2, Max: 0.2, Diff: 0.2, Sum: 1.3]
 [Update RS(ms): Min: 17.0, Avg: 17.2, Max: 17.3, Diff: 0.4, Sum: 137.3]
 [Processed Buffers: Min: 19, Avg: 21.0, Max: 23, Diff: 4, Sum: 168]
 [Scan RS(ms): Min:10.5, Avg:10.7, Max: 10.9, Diff: 0.4, Sum: 85.4]
 [Code Root Scanning(ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
 [Object Copy(ms): Min: 358.7, Avg: 358.8, Max: 358.9, Diff: 0.2, Sum: 2870.3]
 [Termination(ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.7]
 [GC Worker Other(ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
 [GC Worker Total (ms): Min: 386.7, Avg: 386.9, Max: 387.0, Diff: 0.2, Sum: 3095.3]
 [GC Worker End (ms): Min: 277118.8, Avg: 277118.8, Max: 277118.8, Diff: 0.0]
 [Other: 440.0ms]
 [Evacuation Failure: 437.5ms]
 [Choose CSet: 0.0ms]
 [Ref Proc: 0.1ms]
 [Ref Enq: 0.0ms]
 [Redirty Cards: 0.9ms]
 [Humongous Reclaim: 0.0ms]
```

[1] 即 Live Object Graph。

```
   [Free CSet: 0.9ms]
   [Eden: 831.0M(900M)->0.0B(900M) Survivors: 0.0B->0.0B Heap:
1020.1M(1024.0M)->1020.1M(1024.0M)]
   [Time: user=3.64 sys=0.20,real=0.83 secs]
```

按照以下两点执行之后可以避免上述错误。

- 因为 GC 是针对存活对象的回收，而存活对象的回收标准（阈值）和应用程序的性能关系很紧密，且并行 GC 线程和应用线程一起工作，也会让 Java 应用线程[1]过来帮助，所以增加并行线程数量可以减少并行循环运行的时间。回到主题，如果标记阈值较高，可能会导致最终整个并行标记阶段失败。如果过低，会很频繁地触发并行循环，最终导致混合回收“no space”错误。综合来看，最好是在合适的时间开始并行循环，如果做不到这一点，宁可选择早一点进入并行标记循环阶段，也不要选择迟点进入，因为并行循环运行频繁总比发生错误容易处理。
- 假设阈值设置没有问题，但是并行循环依然耗时较长，并且混合回收阶段在声明区间是以“losing the race”这样的字眼失败了，可以通过增大-XX:ConcGCThreads 选项的方式来增加处理线程。这个值的默认值是-XX:ParallelGCThreads 的 1/4，通过修改两个值的任意一个，另一个值也会相应地被修改。注意，增加并行线程的数量会导致 Java 应用程序线程执行更长的时间，因为并行 GC 线程和应用程序线程是在一起执行的。

5.4　混合回收阶段[2]

5.4.1　初步介绍

前面已经介绍了年轻代回收优化、并行标记循环优化，现在要开始进入老年代回收优化，也就是我们所说的混合回收循环优化阶段。一个混合回收 CSet 包括所有的年轻代区间、老年代挑选的若干区间。

优化混合回收会引起 CSet 里的老年代区间的数量变更过程中断，并且增加两次连续的混合回收，这就是所谓的背靠背（back-to-back）混合回收，这样的事件会整合两次分开执行老年区间的耗时。优化混合回收很重要，对 GC 耗时和响应性都有提升。选项-XX:+PrintAdaptiveSizePolicy 会自动下载 G1 的完整生态系统内容（G1's ergonomics heuristic decisions）。

---

[1] 即 Mutator Thread。

[2] 即 Mixed Garbage Collection Phase。

代码清单 5-9　混合回收日志示例片段 1

```
   97.859:[GC pause(G1 Evacuation Pause) (mixed)97.859:[G1Ergonomics((CSet
Construction)start choosing CSet,_pending_cards:28330,predicted base
time:17.45ms,remaining time:182.55ms,target pause time:200.00ms]]
   97.859:[G1Ergonomics(CSet Construction)ass young regions to CSet,eden:37
regions,survivors:14 regions,predicted young region time:16.12ms]
   97.859:[G1Ergonomics(CSet Construction) finish adding old regions to
CSet,reason:old CSet region num reached max,old:103 regions,max:103 regions]
   97.859:[G1Ergonomics(CSet Construction) finish choosing CSet,eden:37
regions,survivors:14 regions,old:103 regions,predicted pause
time:123.38ms,target pause time:200.00ms]
   97.905:[G1Ergonomics(Mixed GCs)continue mixed GCs,reason:candidate old
regions available,candidate old regions:160 regions,reclaimable:66336328
bytes(6.18%),threshold:5.00%],0.0467861 secs]
```

第 1 行告诉我们评估暂停类型，这个日志里面显示的是一个混合回收，预测的时间是针对 CSet 选择和增加年轻代、老年代区间到 CSet。

第 3 行有这样的语言“old CSet region num reached max”出现，可以通过选项 G1OldCSetRegionThresholdPercent 来设置可以添加到 CSet 的老年代区间的数量。

第 5 行（标记为 Mixed GCs）我们可以看到 G1 决定继续混合回收，主要原因是满足了混合回收的两个必要条件，即可用区间和可用堆内存大于默认的 5%的阈值限制。

这个示例最主要的优化是指增加老年代区间到 CSet 结束时间以及持续的混合 GC 阶段回收的空间，我们下面具体介绍这些。

5.4.2　深入介绍

-XX:G1HeapWastePercent 表示最大可以忍受的垃圾总量，它描述了一种你可以容忍的最大垃圾数量的衡量尺度。垃圾也可以理解为碎片[1]，默认是整个 Java 堆空间的 5%（JDK8U45）。

代码清单 5-10　混合回收日志示例片段 2

```
   123.563:[GC pause(G1 Evacuation Pause)(mixed) 123.563:[G1Ergonomics(CSet
Construction)start choosing CSet,_pending_cards:7404,predicated base
time:6.13ms,remaining time:43.87ms,target pause time:50.00ms]
   123.563:[G1Ergonomics(CSet Construction) finish adding old regions to
```

[1] 英文描述主要是 fragmentation。

```
CSet,reason: predicted time is too high, predicted time:0.70ms,remaining
time:0.00ms,old:24 regions,min:24 regions]
   123.563:[G1Ergonomics(CSet Construction) added expensive regions to
CSet,reason:old CSet region num not reached min,old:24 regions,expensive:24
regions,min:24 regions,remaining time:0.00ms]
   123.563:[G1 Ergonomics(CSet Construction) finish choosing CSet,eden:464
regions,survivors:36 regions,old:24 regions, predicted pause time:101.83
ms,target pause time:50.00ms]
   123.640:[ G1 Ergonomics(Mixed GCs) continue mixed GCs,reason:candidate old
regions available,candidate old regions:165 regions,reclaimable:109942200
bytes(10.24%),threshold:5.00%,0.0771342 secs ]
```

最后一行显示垃圾占有量为 10.24%，因为我们设置了回收阈值为 5%，10.24%（109942200/1024/1024=1052MB）大于 5%，所以混合回收会继续进行，这个混合回收过程花费了 77.1ms。

如果混合回收的停顿时间呈现指数级上升，那么可以设置较大的阈值，阈值增大的结果是可能会产生更多的碎片空间，也意味着老年代有更多的瞬时存活数据（transient live data），为了处理这些瞬时存活数据，我们需要调整标记阈值。

-XX:G1MixedGCCountTarget 选项默认值是 8，表示一个混合回收循环阶段内的每次混合回收暂停对应的 CSet 里的老年代区间的数量的最小阈值。相关的完成计算公式为：

```
   Minimum Old CSet Size Per Mixed Collection Pause = Total Number of Candidate
Old Regions Identified for Mixed Collection Cycle / G1MixedGCCountTarget。
```

这个公式决定了每一次混合回收里每一个 CSet 里的最小老年代区间的数量，设置为 X，对应地以老年区间所有的候选人为 Y，都在二次混合收集里。二次混合回收在一个并行标记循环完成后开始。

我们来看一个例子：

```
   123.563:[G1 Ergonomics(CSet Construction)added expensive regions to
CSet,reason:old CSet Region num not reached min,old:24 regions,expensive:24
regions,min:24 regions, remaining time:0.00ms]
```

从上面的 GC 日志可以看出只有 24 个区间被添加到了 CSet，因此没有达到最小老年代区间允许加入到每个 CSet 的上限。再来看一下前面年轻代回收暂停阶段显示有 189 个候选的老年代区间可用，因此 G1 GC 会进入下一个混合收集循环。

```
   117.378:[G1 Ergonomics(Mixed GCs) Start mixed GCs,reason:Candidate old
regiosn available,candidated old regions:189 regions,reclaimable:134888760
bytes(12.56%),threshold:5.00%]
```

我们用 189 除以 8（G1MixedGCCountTarget 的默认值），即 189/8=24，这就是为什么老年代的最小区间数量为 24。

既然有一个加入 CSet 的最小老年代区间阈值，也就会有一个最大值。这个最大值是通过选项-XX：G1OldCSetRegionThresholdPercent 来设置，默认值是整个 Java 堆的 10%。

代码清单 5-11　混合回收日志示例片段 3

```
   97.859:[GC pause(G1 Evacuation Pause)(mixed)97.859:[G1Ergonomics(CSet
Construction) start choosing CSet,_pending_cards:28330,predicted base
time:17.45ms,remaining time:182.55ms,target pause time:200.00ms]
   97.859:[G1Ergonomics(CSet Construction) add young regions to CSet,eden:37
regions, survivor:14 regions, predicted young region time:16.12ms]
   97.859:[G1Ergonomics(CSet Construction) finish adding old regions to
CSet,reason:old CSet region num reached max, old:103 regions, max:103 regions]
   97.859:[G1Ergonomics(CSet Construction) finish choosing CSet, eden:37
regions, survivor:14 regions,old:103 regions,predicted pause
time:123.38ms,target pause time:200.00ms]
   97.905:[G1Ergonomics(Mixed GCs) continue mixed GCs,reason:candidate old
regions available,candidate old regions:160 regions, reclaimable:66336328
bytes(6.18%),threshold:5.00%]
```

从第 3 行和第 5 行我们看到，尽管有更多的可用老年代区间（160 个），在当前 CSet 里面的老年代区间仍然是 103 个，这 103 个区间的数值来自于整个堆大小（1GB）的 10%（选项 G1OldCSetRegionThresholdpercent 设定）。

既然我们知道如何去设置最小和最大的混合回收区的每个 CSet 的最小和最大老年代区间数量，我们就可以修改阈值去适配目标中断时间，以及维持老年代目标存活对象的瞬间数量。

-XX:G1MixedGCLiveThresholdPercent，默认值为 85%（JDK8U45），设置一个 CSet 里每一个区间最多存活对象的百分比，这个百分比值是在并行标记阶段计算出来的。

存活对象百分比大于活性阈值以上的老年代区间没有被包含在 CSet 候选区间里面，这个选项直接决定了每个区间的碎片率。注意，每次回收时间也会增长，最终导致混合回收暂停耗时增长。

5.5　如何避免出现 GC 失败

前面这个章节我们讨论了几种避免回收失败的方法，现在我们讨论 4 个和优化相关的重要选项。

- Java 堆大小。确保可以存放所有的静态和瞬时存活对象，以及短期、中期应用程序存活对象。除了存活对象空间以外，尽可能地增加一些堆空间，这样可以应对突发事件。预留空间有助于 GC 操作，预留空间越多，GC 的吞吐量越高，应用程序的延迟越低。
- 避免过多地配置 JVM 命令行选项。对于你不熟悉的选项，尽量用默认值。一般用户只需要设置初始化堆大小和最小堆大小，例如-Xms2g –Xmx4g –XX:MaxGCPauseMills=100 –XX:InitatingHeapOccupanyPercent=55。
- 如果应用程序有很多大对象，且长时间存在，要确保标记阈值设置较低，并且确保大对象的划分界限正确。-XX:G1HeapRegionSize 选项设置了一个值，对应为一个区间大小的 50%以上，确保为大对象，区间大小为 1MB 到 32MB，我们来看一个示例，如代码清单 5-12 所示。

代码清单 5-12　大对象日志示例

```
91.890:[G1Ergonomics(Concurrent Cycles) request concurrent cycle initiation,
reason:occupancy higher than threshold,occupancy:483393536 bytes,allocation
request:2097168  bytes,threshold:483183810  bytes(45.00%),source:concurrent
humongous allocation]
```

并行循环启动是由于堆内存保有率被突破了，进一步的原因是大对象内存请求。请求数量为 2097168bytes，大于 2MB，已经突破了默认的 1MB/区间（JVM 启动时设置）大小。

- 如果评估失败[1]发生了几次，且都是因为幸存者区间没有足够的空间用于新的升级对象，你可以增大选项-XX:G1ReservePercent。这就如同天花板效应，天花板一直在那里，但人和天花板之间的距离可以有大有小。其默认值为 Java 堆空间的 10%，最大为 50%。

5.6　引用处理

GC 对于不同的引用对象类型有不同的处理方式，很明显，在回收过程中，GC 花费在软引用、弱引用、虚引用上的工作步骤和难度要大于强引用[2]。

这一章会告诉我们如何证明 G1 回收阶段在处理引用过程中花费了过多的时间，如何解决 G1 处理引用过程的性能瓶颈，也许需要你重构应用程序的代码。

---

[1] 国外 GC 文章一般为 Evacuation Failures。

[2] 即 Strong References、Soft References、Weak References、Phantom References。

5.6.1 观察引用处理

G1 GC 在处理引用对象时按照的是发现引用对象、推入 JVM 的引用队列、从引用队列拿出对象、处理对象这四个步骤。我们在 G1 GC 的打印日志里面可以发现入队时间和处理时间，这两个时间都在年轻代和混合回收的“Other”这一栏里面显示，如代码清单 5-13 所示。

代码清单 5-13 观察引用处理

```
[Other:9.9ms]
   [Choose CSet:0.0ms]
   [Ref Proc:8.2ms]
   [Ref Enq:0.3ms]
   [Redirty Cards:0.7ms]
   [Humongous Reclaim:0.0ms]
   [Free CSet:0.5ms]
```

“Ref Proc”描述的是处理引用对象花费的时间，而“Ref Enq”是引用对象出队列花费的时间。从上面的例子可以看到，花费在“Ref Enq”的时间比“Ref Proc”少，大多数应用程序都是这种情况。

G1 也会在并行循环的重标记阶段打印引用处理细节，如下所示：

```
[GC remark[Finalize Marking,0.0007422 secs][GC ref-proc,0.0129203
secs][Unloading,0.0160048 secs],0.0308670 secs]
```

-XX:+PrintReferenceGC 选项打印每一个循环内部的每一个引用对象细节，这个选项对于找到花费最长时间的引用对象很有用。示例如下所示。

```
[GC remark[Finalize Marking,0.0003322 secs][GC ref-proc[SoftReference,2 refs,
0.0052297 secs][WeakReference,3264308 secs,2.00524238 secs][FinalReference,
215 refs,0.0028225 secs][PhantomReference,1787 refs,0.005004 secs][JNI Weak
Reference,0.00777 secs],2.0932151 secs][Unloading,0.0060031 secs],2.120140
secs]
```

G1 GC 在选项-XX:+PrintGCDetails 里面就可以打印引用处理时间，其他 HotSpot GC 必须使用-XX:+PrintReferenceGC 选项才能引用这些信息。

如果“Ref Proc”阶段花费的时间大于整个 GC 暂停阶段的 10%，我们就需要对年轻代和混合 GC 阶段的引用对象进行优化了。G1 的重标记阶段很容易发现引用处理的时间是否占用了大量的回收暂停时间，这要从重标记阶段的功能聊起。并行循环的重标记阶段开始于大量的引用对象在老年代回收阶段被发现之后，既然发现了它们，那么就会在重标记阶段处理它们，由于

是批量发现的，不是单一的引用对象，所以处理时间一定也需要较长。如果重标记阶段处理引用对象的时间很长，甚至超过了最大暂停目标时间，或者说它是耗时最长的处理阶段，那么我们就需要做相应的优化了。

5.6.2　引用处理优化

所有的优化手段都是差不多的。首先，我们想到的是在默认情况下 HotSpot 处理引用对象是单线程方式，即在这种默认状态下使用最小化内存、让出 CPU 资源给其他应用程序，这种无私的精神当然意味着处理引用对象的耗时较大，那么我们可以通过选项-XX:+ParallelRefProcEnabled 开启多线程处理，这样处理引用对象时虽然会消耗更多的 CPU 资源，但是会缩短耗费在引用处理阶段的系统耗时。

如果启用了-XX:+ParallelRefProcEnabled 选项之后引用处理时间依然大于年轻代或者混合 GC 的暂停时间的 10%，或者 G1 的重标记阶段依然花费了较多的时间在引用处理上面，那么我们下一步是找到哪一个引用对象类型或者引用对象类型集合耗费了这么长的时间。我们可以通过选项-XX:+PrintReferenceGC 打印每一个引用对象类型的统计信息，如下所示：

```
[GC pause(young)[SoftReference,0 refs,0.0001139 secs][WeakReference,26845
refs,0.0050601 secs][FinalReference,5216 refs,0.0032409 secs][PhantomReference,
336 refs,0.0000986 secs][JNI Weak Reference,0.0000837 secs],0.0726876 secs]
```

如果我们看到一个特定引用对象类型耗时很长，那可能意味着应用程序过度地使用了它。如下所示，弱引用花费的时间（2s 左右）远远大于其他引用类型花费的毫秒级别的时间。

```
[GC remark[Finalize Marking,0.0003322 secs][GC ref-proc [SoftReference,2
refs,0.005296 secs][WeakReference,3264308 refs,2.0524238 secs][FinalReference,
215 refs,0.0028225 secs][PhantomReference,1787 refs,0.0050046 secs][JNI Weak
Reference,0.0007776 secs],2.093215 secs][Unloading, 0.0060031 secs],2.1201401
secs]
```

在上面的示例中弱引用处理时间很长，那么我们可以通过对应用程序代码重构来减少弱引用处理的负荷，或者减少引用对象类型的回收时间。一般情况下处理该问题需要两个步骤。

1. 验证选项-XX:+ParallelRefProcEnabled 是否启用。如果没有，启用这个选项并且观察是否减少了暂停时间，该暂停时间应该达到我们的暂停目标。

2. 如果选项-XX:+ ParallelRefProcEnabled 启用，告诉程序员你发现了很多弱引用对象并且处理时间很长，请他们查看代码并做重构，减少对于弱引用对象的引用。

软引用处理耗时比较长的情况我们也应该重点关注。如果我们通过日志观察到软引用耗时

较长，我们需要进一步观察是否老年代回收循环比较频繁，因为老年代回收循环由并行循环组成，后面会跟着一系列的混合回收事件。如果你看到很多的软引用被处理并且 GC 事件发生很频繁，或者看到堆内存的使用率一直在最大堆内存附件徘徊，那么就启用-XX:SoftRefLRUPolicyMSPerMB 选项回收软引用对象，该选项默认值是 1000ms，意味着软引用对象会在 Java 堆内部空闲时间大于 1000ms 以上时被回收。这句话有点绕口，我们举一个例子，假设现在有 1GB 的空闲空间，也就是说 1024MB，任何软引用对象有超过 1024×1000=1024000ms，或者说是 1024s（约 17 分钟）没有被访问了，那么该软引用对象就会被 HotSpot 垃圾回收器清除和回收。

-XX:SoftRefLRUPolicyMSPerMB 选项如果设置了一个比较小的值，那么意味着软引用对象会被频繁回收，这样会导致 GC 事件之后较小的堆占用率，或者换句话说，更少的存活数据。反之就是更多的存活数据。

5.7 本章小结

本章对前一章介绍的年轻代、老年代、混合回收、并行标记阶段等基本概念进行了深入梳理，重点介绍优化方案，以及提出如何避免出现 GC 失败的诸多方法，此外，也对堆内存对象内部存在的引用处理机制进行了解释。

6 第6章 JVM诊断工具使用介绍

学西医的人普遍认为神经学是医学领域里较难的一个细分领域。由上千亿个神经细胞和1014个以上的突触组成的人类大脑及周围神经系统具有极为复杂精细的结构和功能。由脑、脊髓组成的中枢神经系统和由脑神经、脊神经组成的周围神经系统组成了一个完整、统一、和谐的整体，指挥和协调躯体的运动、感觉和自主神经功能，感受机体内外环境传来的信息并做出反应，参与人的意识、学习、记忆、综合分析等高级神经活动。神经病学（neurology）作为从内科学中派生出来的学科，是研究中枢神经系统、周围神经系统及骨骼肌疾病的病因、发病机制、病理、临床表现、诊断、治疗及预防的一门临床医学门类。追踪Java应用程序问题最难的是JVM底层状态和问题探究，定位难度和神经学研究方向类似。

我们通常都是对Java应用层进行问题追踪、调试，但是很少关注JVM层级的问题查找。Serviceability Agent工具主要提供了一种调试存活Java进程的能力，或者更加形象一点，它能够读懂Java应用程序崩溃之后生成的Crash Dump File[1]。

[1] 包括一个程序的完整信息，比如内存状态、进程注册状态、堆栈信息、其他进程信息、操作系统标记信息等。

总的来说，SA 工具对于定位应用程序停顿、内存泄漏、程序崩溃、不符合常规理解的错误等严重问题很有效。

本章主要介绍和解决以下问题，这些主要针对 SA 工具进行学习。

- 什么是 Serviceability Agent 工具？
- 了解相关概念、工具的基本功能及实践介绍。
- 了解工具的扩展功能、插件介绍。
- 对一些常见问题作出总结。

6.1 SA 基础介绍

我们已经有了很多工具可以用来跟踪、调试 Java 应用程序出现的信息或者异常问题，比如 JConsole、JMap 等，但是很少有工具是针对 JVM 层级的，这个领域太靠近底层，有很多函数不是用 Java 语言实现的，很多情况需要调用本地函数（C 语言编写），所以了解和实现的难度比较大。本章着重介绍的 Serviceability Agent（SA）是一个调试工具集合[1]，它可以针对 Java 应用程序，也可以针对 JVM 问题进行调试。SA 提供了调试 Java 进程的能力，同时也提供了自动分析 Crash Dump Files（异常崩溃文件）的能力。

我们查找高性能应用程序出现异常的一般思路是能够快速发现并明确解决问题[2]，通常这些问题包含应用程序挂起、内存泄漏（Memory Leaks）、未知错误、JVM 崩溃等。通过 SA 可以帮助我们构建稳定、可靠、可扩展以及高效的 Java 应用程序。

通俗点讲，Serviceability Agent 是 Sun 公司的 HotSpot 内部调试代码的集合体，已经被用在了 jstack、jmap、jinfo、jdb 这些工具的编写上面。[3]

SA 工具在 JDK 的 lib 目录下面，是一个 jar 文件，如果你把这个 jar 文件执行解压缩操作，你会发现它由 com、META-INF、sun、toolbarButtonGraphics 这四个文件夹组成，另外包含一个 sa. properties 文件。这四个文件夹的对应作用如下。

- Com 文件夹存放的是 GUI 相关的类文件。

---

[1] 英文原文为 Set of debugging tools。

[2] 英文原文为 TroubleShooting 和 Nailing Down。

[3] 英文原文 The Serviceability Agent is collection of Sun internal code that aids in debugging HotSpot problems. It is also used by several JDK tools - jstack, jmap, jinfo, and jdb. See SA for more information。

- Sun 文件夹存放的是服务端程序对应的类文件。
- META-INF 文件夹存放的是内部 JDI(Java Debug Interface)，用于定义调试器(Debugger)所需要的一些调试接口。
- toolbarButtonGraphics 文件夹存放的是图标文件。

sa.properties 文件里面存放的是一个版本号，我们使用的是 jdk1.8.0_51，对应的内容为 sun.jvm.HotSpot.runtime.VM.saBuildVersion=25.51-b03。为什么需要这个版本号呢？因为 HotSpot 是采用 C++语言编写的，SA 工具包是采用 Java 语言编写的，而 SA 工具包又是针对 HotSpot 的一个调试工具（魔镜），所以它的代码、方法需要和 HotSpot 保持一致，这个版本号就是起这个作用，便于两者对应版本。如果版本不一致，会在调用 SA 的 jar 包时抛出 VMVersion-MismatchException 错误。当然，这个检查也是可以忽略的，通过在调用 SA 的 jar 包时指定选项 Dsun.jvm.HotSpot.runtime.disableVersionCheck 的方式来关闭自动检查。

6.2　SA 工具使用实践

6.2.1　如何启动 SA

SA 中的包主要分为以下几个组件，即 asm、ci、code、debugger、gc、interpreter、jdi、livevm、memory、oops、opto、prims、runtine、tools，以及 types 等。

JDK 提供了两种调试工具，这两种工具都使用了 Serviceability Agent 的 API 接口，它们分别是 HSDB 和 CLHSDB[1]。两者的区别是 HSDB 提供了丰富的图形化界面支持，而 CLHSDB 提供的是命令行式的功能。对于调试本地进程，HSDB 比较有优势，毕竟有界面，但是对于调试远程服务器上的问题，特别是核心崩溃文件[2]的解析，用命令行比较方便。

6.2.1.1　如何启动 HSDB

1. 安装 JDK，如图 6-1 所示，这个步骤比较简单，Windows 操作系统只要按照 exe 程序的顺序进行安装即可，Linux 操作系统读者可以搜索百度，有很多帖子介绍如何安装，这里就不多涉及。

---

[1] 全称分别是 HotSpot Debugger 和 Command-Line HotSpot Debugger。

[2] Core dump file。

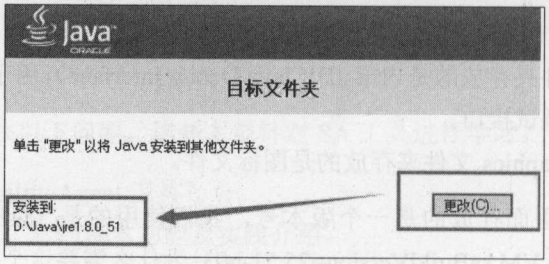


图 6-1　安装 JDK8 的运行界面

2. 设置 JAVA_HOME，只有设置了环境变量才能让应用程序找到 JDK 程序，如图 6-2 所示。

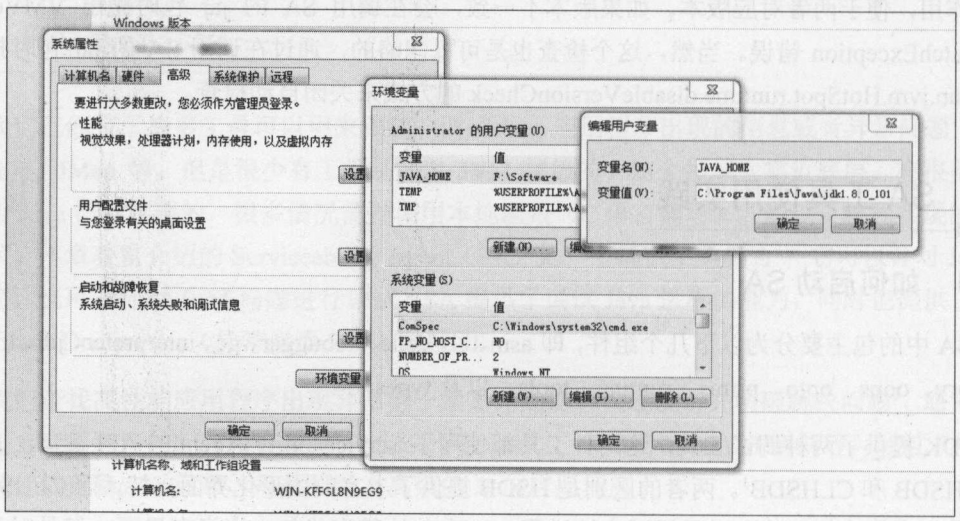


图 6-2　设置 JAVA_HOME 环境变量

3. 设置 PATH 变量，如图 6-3 所示。

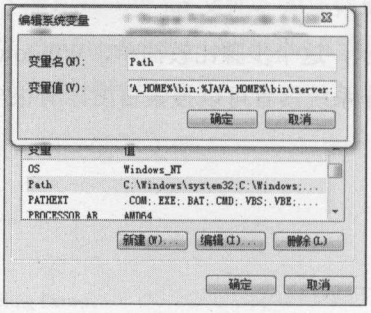


图 6-3　设置程序路径

4. 执行命令

平时用来查看 HotSpot 内部信息的常用工具都在$JAVA_HOME/bin 目录下，其中一些工具就是用 Serviceability 开发的。

6.2.1.2　Linux 运行命令方式

运行命令，启动 HSDB 工具的图形化界面。Linux 操作系统的命令是 Java –classpath $JAVA_HOME/lib/sa-jdi.jar sun.jvm.HotSpot.HSDB。如果想要启动 command 命令行模式，可以使用命令 Java -classpath $JAVA_HOME/lib/sa-jdi.jar sun.jvm.HotSpot.CLHSDB。

如果出现这样的错误：Error: Could not find or load main class sum.jvm.HotSpot.HSDB，请查看 JAVA_HOME、Classpath 的设置是否正确。

Linux Centos7.0 环境下的设置方式是在/etc/profile 文件最末端加入可用的 jdk 路径，如代码清单 6-1 所示。

代码清单 6–1　设置/etc/profile 方法

```
export JAVA_HOME=/usr/lib/jdk1.7.0_79
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/lib:$JRE_HOME/bin:$PATH
```

然后 source /etc/profile，接下来再执行命令就可以了。

6.2.1.3　Windows 运行命令行方式

```
java -cp .;%JAVA_HOME%/lib/sa-jdi.jar sun.jvm.hotspot.HSDB
```

6.2.1.4　如何启动 CLHSDB

回到 CLHSDB，这个只是一个壳子，主要的功能都是靠 sun.jvm.HotSpot.HotSpotAgent 和 sun.jvm.HotSpot.CommandProcessor 完成的。运行一个示例：Java -classpath $JAVA_HOME/lib/sa-jdi.jar sun.jvm.HotSpot.HelloWorld，程序的输出为“HelloWorld.d() received "Hi" as argument Going to sleep...”。

HelloWorld 的源代码如代码清单 6-2 所示。

代码清单 6–2　HelloWorld 源代码

```
packagecom.hikvision.ivms.server;
```

```
importJava.lang.reflect.Method;

publicclass HelloWorld {
    privatestatic String helloWorldString = "Hello, world!";
    privatestaticvolatileinthelloWorldTrigger = 0;
    privatestaticfinalbooleanuseMethodInvoke = false;
    privatestatic Object lock = new Object();

    publicstaticvoid main(String[] args) {
    intfoo = a();

    System.out.println("HelloWorld exiting. a() = " + foo);
      }

    privatestaticint a() {
    return 1 + b();
      }

    privatestaticint b() {
    return 1 + c();
      }

    privatestaticint c() {
    return 1 + d("Hi");
      }

    privatestaticint d(String x) {
    System.out.println("HelloWorld.d() received \"" + x + "\" as argument");
    synchronized(lock) {
    if (useMethodInvoke) {
    try {
            Method method = HelloWorld.class.getMethod("e");
            Integer result = (Integer) method.invoke(null, new Object[0]);
    returnresult.intValue();
          }
    catch (Exception e) {
    thrownewRuntimeException(e.toString());
          }
        } else {
```

```
inti = fib(10); // 89
longl = i;
floatf = i;
doubled = i;
charc = (char) i;
shorts = (short) i;
byteb = (byte) i;

intret = e();

System.out.println("Tenth Fibonacci number in all formats: " +
i + ", " +
l + ", " +
f + ", " +
d + ", " +
c + ", " +
s + ", " +
b);

returnret;
    }
  }
 }

publicstaticint e() {
System.out.println("Going to sleep...");

inti = 0;

while (helloWorldTrigger == 0) {
if (++i == 1000000) {
System.gc();
    }
  }

System.out.println(helloWorldString);

while (helloWorldTrigger != 0) {
  }

returni;
```

```
    }

    // Tree-recursive implementation for test
    publicstaticint fib(intn) {
    if (n< 2) {
    return 1;
      }
    returnfib(n - 1) + fib(n - 2);
    }

}
```

6.2.2 SA 原理及使用介绍

大家是否记得韩剧《来自星星的你》，外星人可以让时间停滞，在这个停滞时间段里可以做很多很多事情。SA 就是这么一款由一系列 Java API 组成的工具，当它工作时，对应的 Java 应用程序进程就停下来了[1]，随后 SA 开始检查 Java 堆内的内容、线程执行情况、HotSpot VM 的内部数据结构、加载的类、方法区等，等检查完毕后对应的 Java 应用程序进程恢复运行。所以这个过程有点像一个瞬时的冰封时代，采用英文描述可以说 SA 是一种 SnapShot Debugger Tool（通常情况下的调试工具大多是采用针对应用程序进程的单步调试方式，即 Step Through A Running Program）。

使用 SA 的理由如下所述。

- 如果你想要跨平台的工具，那就用 SA 吧。
- 如果你想要读懂操作系统、JVM 抛出的十六进制信息，就用 SA 吧。

6.2.2.1 绑定（Attach）本地 HotSpot 进程

如果你需要查看本地 JVM 进程的执行情况，我建议你绑定本地 HotSpot 进程，首先查看进程 ID 号码，即 PID。我们可以在任务调度器（Windows 环境下）找到“javaw.exe”这个进程，默认没有输出 PID，如图 6-4 所示。

[1] 即挂起，halted。

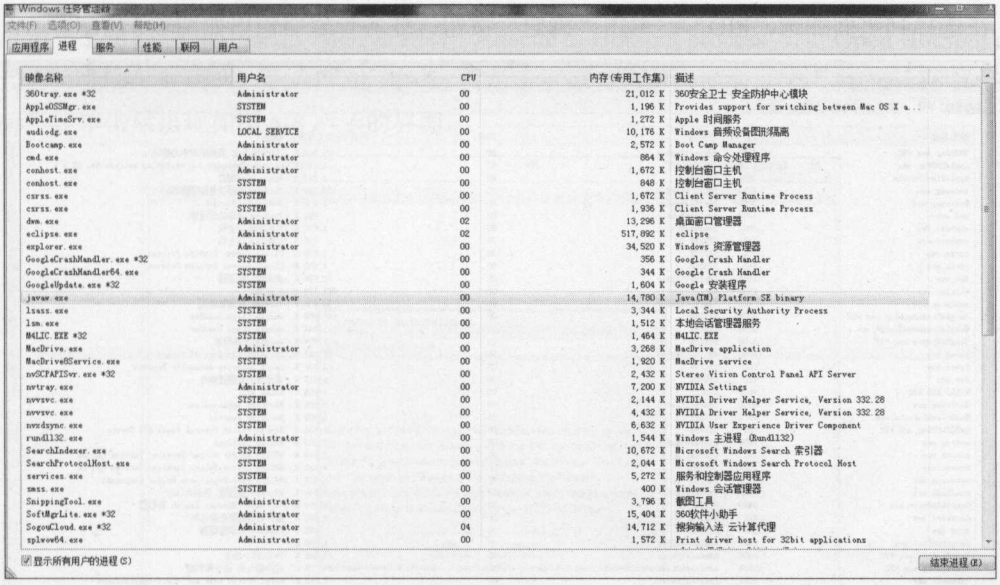


图 6-4　如何查看进程 ID 号码

接着我们可以增加任务管理器的显示字段，这里需要增加 PID 显示，我们单击任务管理器的“查看-选择列”，如图 6-5 所示。

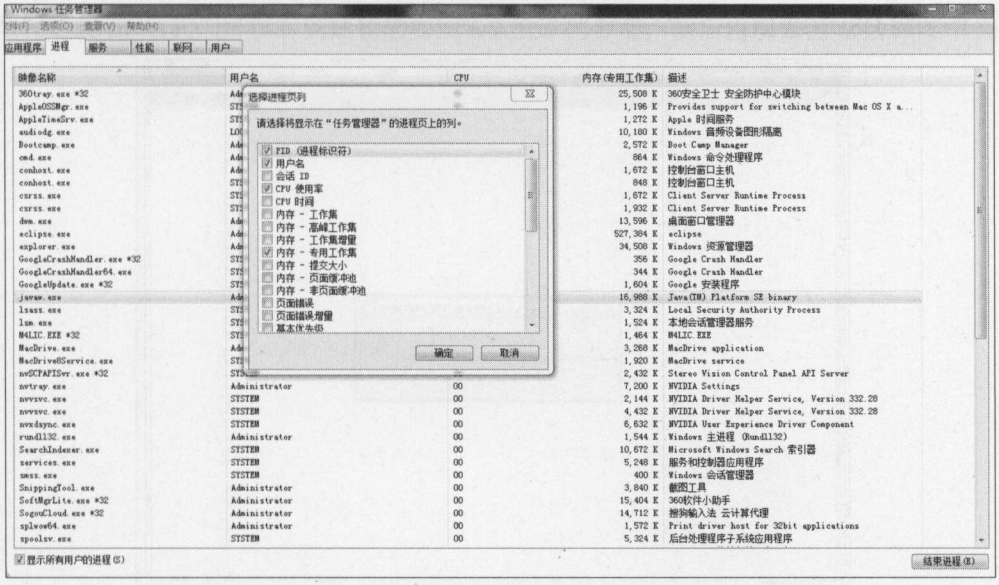


图 6-5　如何增加显示列

在选择列里面加选“PID”，单击“确定”按钮，任务管理器输出如图 6-6 所示。

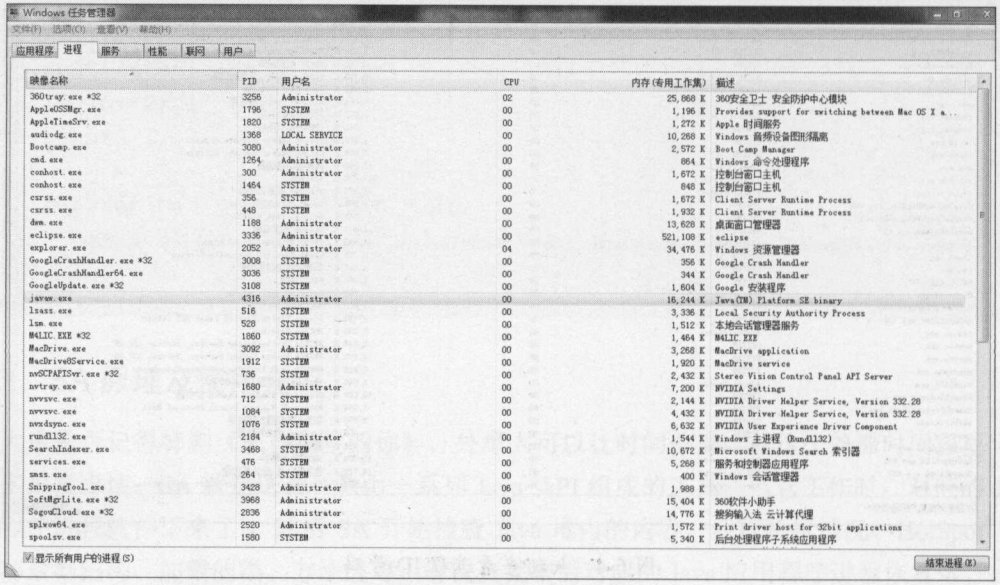


图 6-6　增加显示列 PID

运行命令：java -cp .;%JAVA_HOME%/lib/sa-jdi.jar sun.jvm.hotspot.HSDB，然后可以看到如图 6-7 所示的 HSDB 界面。

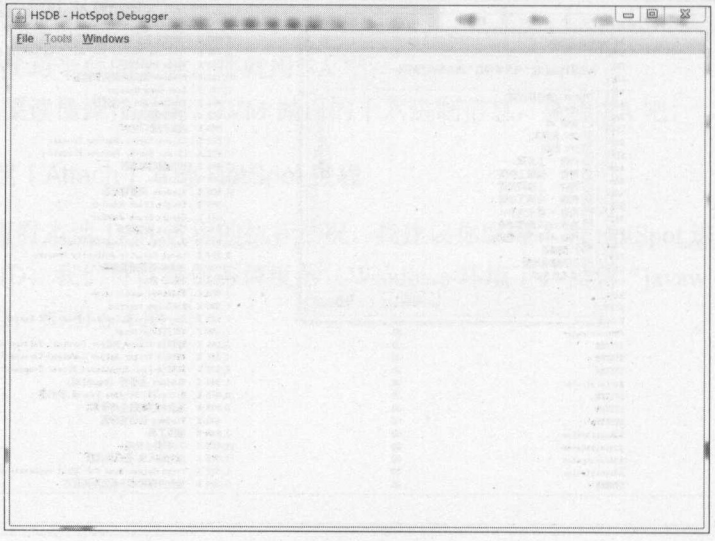


图 6-7　HSDB 初始界面

单击左上角的“File”按钮，你会看到有“Attache to HotSpot process”、“Open HotSpot core file”、“Connect to debug server”这三个选项，这里需要选择的是第 1 个“Attache to HotSpot process”，单击后出现如图 6-8 所示的界面。

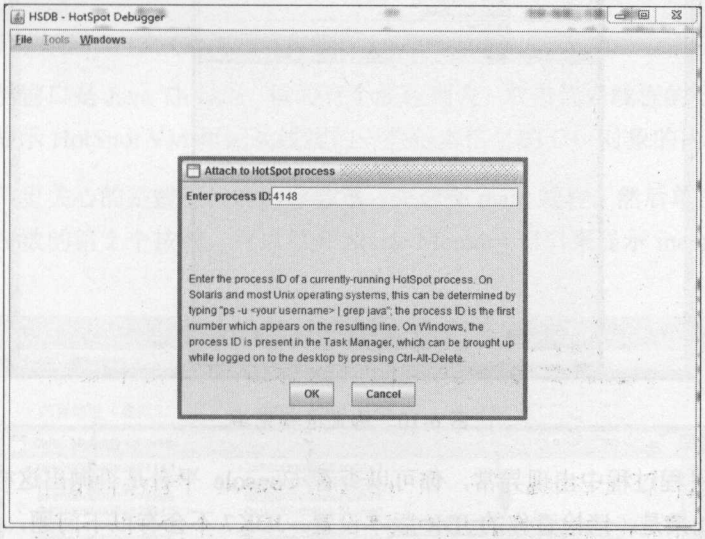


图 6-8　HSDB 填写进程号界面

填写进程 ID 后我们可以看到短暂的如图 6-9 所示的界面，然后出现如图 6-10 所示界面。

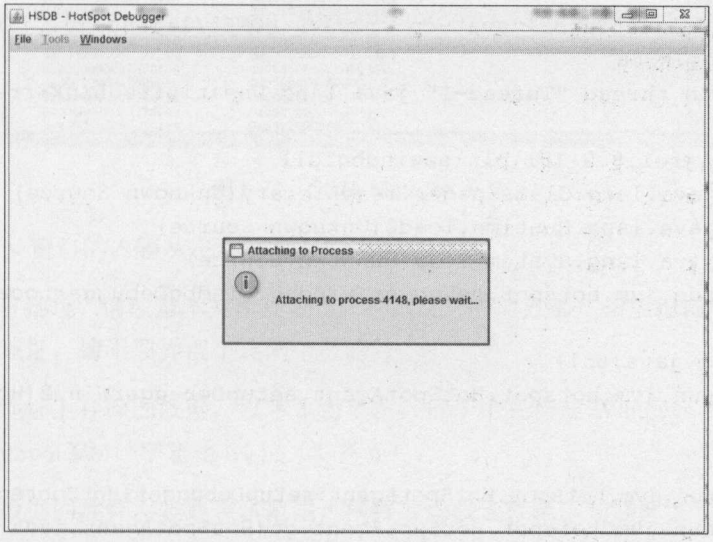


图 6-9　绑定进程过程中

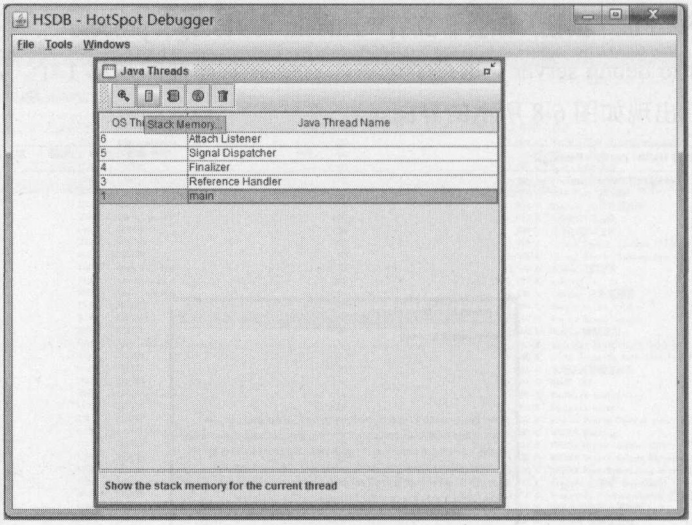


图 6-10　绑定进程完毕

如果在绑定进程过程中出现异常，你可以查看 Console 平台是否抛出这样的错误，如代码清单 6-3 所示。如果是，请检查你的 JDK 版本设置，JDK7 不会有以下问题，JDK8 会出现这个问题。

代码清单 6-3　HSDB 绑定进程异常提示

```
   C:\Users\Administrator>java -cp .;%JAVA_HOME%/lib/sa-jdi.jar
sun.jvm.hotspot.HSDB
   Exception in thread "Thread-1" java.lang.UnsatisfiedLinkError: Can't load
librar
   y: C:\Java\jre1.8.0_101\bin\sawindbg.dll
          at java.lang.ClassLoader.loadLibrary(Unknown Source)
          at java.lang.Runtime.load0(Unknown Source)
          at java.lang.System.load(Unknown Source)
          at sun.jvm.hotspot.debugger.windbg.WindbgDebuggerLocal.<clinit>
(WindbgDe
   buggerLocal.java:651)
          at sun.jvm.hotspot.HotSpotAgent.setupDebuggerWin32(HotSpotAgent.
java:521
   )
          at sun.jvm.hotspot.HotSpotAgent.setupDebugger(HotSpotAgent.java:336)
          at sun.jvm.hotspot.HotSpotAgent.go(HotSpotAgent.java:313)
          at sun.jvm.hotspot.HotSpotAgent.attach(HotSpotAgent.java:157)
```

```
        at sun.jvm.hotspot.HSDB.attach(HSDB.java:1168)
        at sun.jvm.hotspot.HSDB.access$1700(HSDB.java:53)
        at sun.jvm.hotspot.HSDB$25$1.run(HSDB.java:436)
        at sun.jvm.hotspot.utilities.WorkerThread$MainLoop.run(WorkerThread.
java:66)
        at java.lang.Thread.run(Unknown Source)
```

刚开始打开的窗口是 Java Threads，里面有个线程列表。双击代表线程的行会打开一个 Oop Inspector 窗口，显示 HotSpot VM 里记录线程的一些基本信息的 C++对象的内容。

不过这里我们更关心的是线程栈的内存数据。先选择 main 线程，然后单击 Java Threads 窗口里的工具栏从左数的第 2 个按钮，可以打开 Stack Memory 窗口来显示 main 线程的栈，如图 6-11 所示。

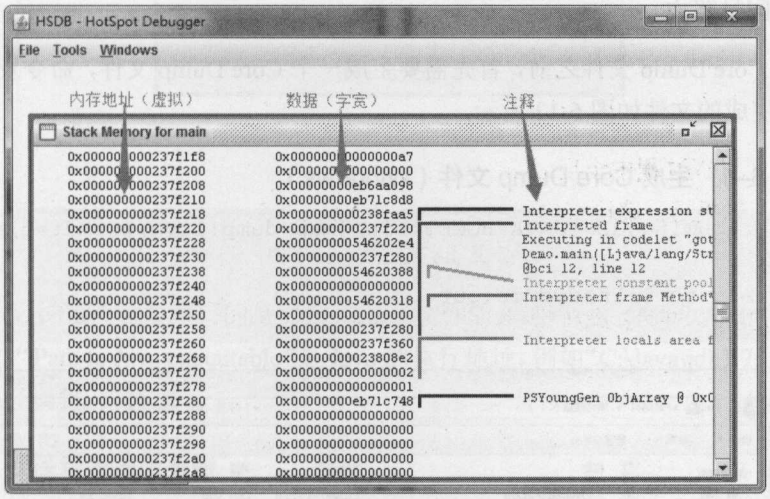


图 6-11　Stack Memory 窗口

Stack Memory 窗口的内容从左到右有三栏。

第 1 栏是内存地址，请注意本文里提到“内存地址”的地方都是指虚拟内存意义上的地址，而不是物理内存地址，请不要弄混了这两个概念。

第 2 栏是该地址上存储的数据，以字宽为单位，例子中我是在 Windows 7 64-bit 上运行 64 位的 JDK7 的 HotSpot VM，字宽是 64 位（8 字节）。

第 3 栏是对数据的注释，竖线表示范围，横线或斜线连接范围与注释文字。

当我们绑定这个应用程序进程的时候，该应用程序进程是挂起的，即不会动的、暂停了、

冰冻了。我们拿到数据后可以解绑这个进程。单击 detach 按钮之后，原有界面就消失了，Java 进程又恢复了自己的运行。

6.2.2.2 绑定 HotSpot Core 文件

SA 工具对于事后分析也是很有用的。比如我们可以分析一个 JVM 崩溃时生成的 Core File，HotSpot 的 core dump 文件在 Windows 操作系统里对应的是崩溃日志文件，一个 core dump 文件本身是一个二进制文件（因此我们读不懂的），它的主体内容是特定时间的程序运行状态，有点类似于飞机的黑匣子。Core File 一般的生成时间是在进程崩溃，针对运行中的应用程序进行离线调试的时候。

注意，Core Dump 文件可能会很大，因为它包含了一个时间段内的状态信息，所以我们需要确保磁盘空间足够大。

绑定一个 Core Dump 文件之前，首先需要生成一个 Core Dump 文件，命令及输出如代码清单 6-4 所示，生成的文件如图 6-12 所示。

代码清单 6–4　生成 Core Dump 文件（Windows）

```
   C:\Users\zhoumingyao>%JAVA_HOME%/bin/jmap-dump:live,format=b,file=c:/hea
mdump.out 6096
   Dumping heap to C:\heamdump.out ...
   File created
```

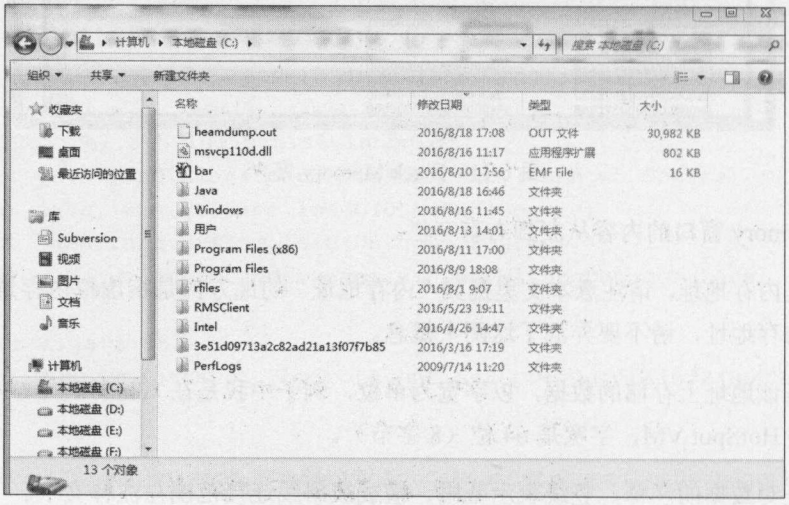


图 6-12　core dump 文件生成

然后我们可以尝试使用 SA 打开 Core Dump 文件，单击“File-Open HotSpot Core File”按钮，弹出对话框如图 6-13 所示。

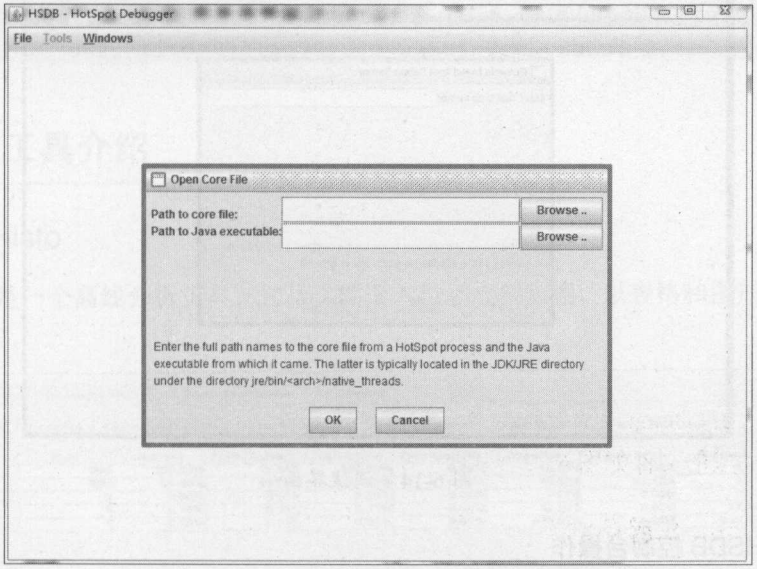


图 6-13　绑定 Core 文件

“Path to core file”需要填写的是 Core Dump 文件的详细路径，你可以单击“Browse”按钮完成文件寻找。“Path to Java executable”是 Java 的执行地址，例如“C:\Java\jdk1.7.0_80\bin\Java”，视自己安装路径调整。

6.2.2.3　连接到调试服务器

这个特性适合于调试远程服务器上的进程或者 Core Dump 文件。我们需要在远程服务器上执行以下步骤，这样就可以启动调试服务器了。

1. 使用 sa-jdi.jar 进行 RMI 注册，执行命令$JAVA_HOME/bin/rmiregistry –J-Xbootclasspath/p:${JAVA_HOME}/lib/sa-jdi.jar。这个命令会创建和开始一个远程对象注册，默认的端口是 1099。

2. 在远程服务器上启动调试服务，需要指明特定的进程号或者 Core Dump 文件。命令为${JAVA_HOME}/bin/java –classpath $JAVA_HOME/lib/sa-jdi.jar sun.jvm.hotspot.DebugServer <pid> [uniqueID]。其中 uniqueID 是可选的，主要用于一台远程服务器上有多个调试服务时，各个服务之间的区别。

3. 连接到 HotSpot 调试服务器，选择“File-Debug Server”，出现如图 6-14 所示的界面。

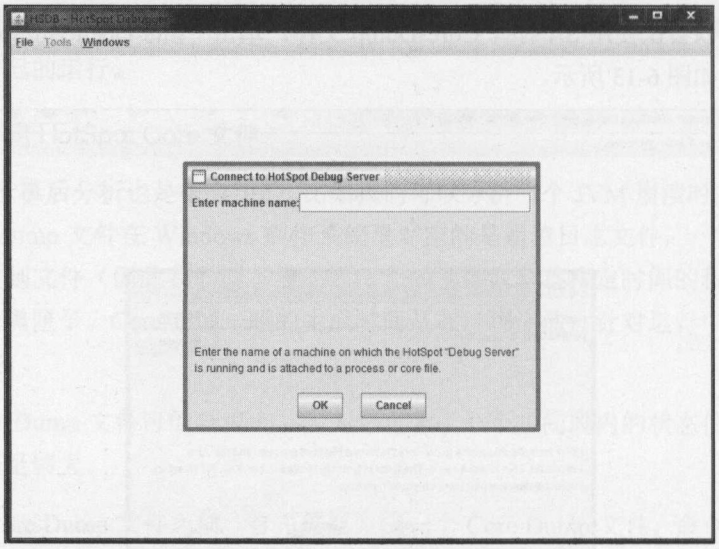


图 6-14　调试界面

6.2.2.4　HSDB 控制台操作

我们可以打开 HSDB 里的控制台，以便用命令来了解更多信息。在菜单里选择 Windows→Console，如图 6-15 所示。

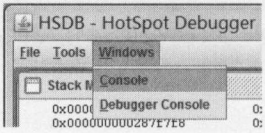


图 6-15　控制台操作

然后会得到一个空白的 Command Line 窗口。在里面敲一下回车就会出现 hsdb>提示符。可以用 universe 命令来查看 GC 堆的地址范围和使用情况，如代码清单 6-4 所示。

代码清单 6-4　universe 命令运行结果

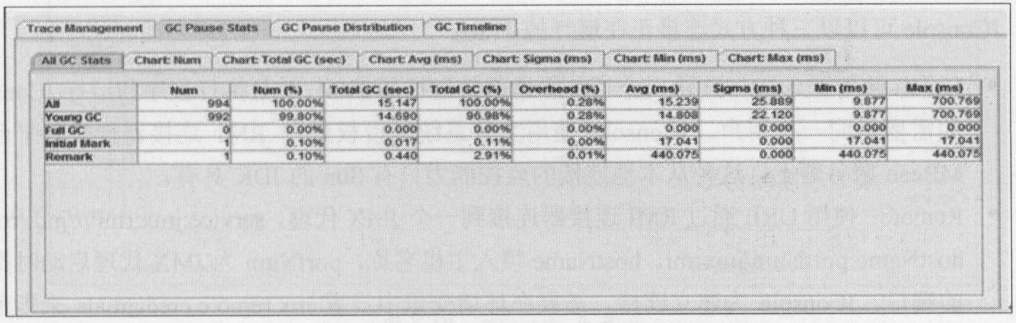
```
   hsdb> universe
   Heap Parameters:
   Gen 0:   eden [0x00000000fa400000,0x00000000fa4aad68,0x00000000fa6b0000)
space capacity = 2818048, 24.831088753633722 used
     from [0x00000000fa6b0000,0x00000000fa6b0000,0x00000000fa700000) space
capacity = 327680, 0.0 used
     to   [0x00000000fa700000,0x00000000fa700000,0x00000000fa750000) space
```

```
capacity = 327680, 0.0 usedInvocations: 0
   Gen 1:   old  [0x00000000fa750000,0x00000000fa750000,0x00000000fae00000)
space capacity = 7012352, 0.0 usedInvocations: 0
   perm [0x00000000fae00000,0x00000000fb078898,0x00000000fc2c0000) space
capacity = 21757952, 11.90770160721009 usedInvocations: 0
```

6.3　其他工具介绍

6.3.1　GCHisto

GCHisto 是一个离线分析工具，它从文件读入垃圾收集数据，以表格和图形化的方式展现，如图 6-16 所示。



	Num	Num (%)	Total GC (sec)	Total GC (%)	Overhead (%)	Avg (ms)	Sigma (ms)	Min (ms)	Max (ms)
All	994	100.00%	15.147	100.00%	0.28%	15.239	25.889	9.877	700.769
Young GC	992	99.80%	14.690	96.98%	0.28%	14.808	22.120	9.877	700.769
Full GC	0	0.00%	0.000	0.00%	0.00%	0.000	0.000	0.000	0.000
Initial Mark	1	0.10%	0.017	0.11%	0.00%	17.041	0.000	17.041	17.041
Remark	1	0.10%	0.440	2.91%	0.01%	440.075	0.000	440.075	440.075

图 6-16　GCHisto 剪切图

GC Pause Stats 页显示了垃圾收集的次数、开销和持续时间等，它的子选项卡逐项集中展示这些数据。所有引入 Stop-The-World 停顿的垃圾收集，在表中都会占用一行，最上面一行是总计。

垃圾收集的开销（Overhead%）表示垃圾收集调优的程度。作为一般性准则，并发垃圾收集的开销应该小于 10%，也有可能达到 1%～3%。对 Throughput 收集器来说，如果垃圾收集的开销接近 1%，说明垃圾收集器调优得很好，3%或更高则说明还可以通过调优改善应用的性能。重要的是理解垃圾收集开销和 Java 堆大小之间的关系，Java 堆越大，降低垃圾收集开销的机会越大。对于给定的 Java 堆大小，通过 JVM 调优才能达到最小的开销。

最长停顿时间（Maximum Pause Time），可以用来评估最差情况下垃圾收集的延时是否满足要求。最长停顿时间超过应用需求时，JVM 可能需要调优，至于是否有必要，则由其超过的程度和超出的停顿时间决定。

默认时，GC Pause Distribution 显示所有垃圾收集停顿的分布，*x* 轴是垃圾收集的停顿时间，*y* 轴是停顿的次数。单独查看 Full GC 通常更有用，因为一般来说它的时间最长。只看 Young GC 可以了解停顿时间的变化分布，停顿时间分布广，说明对象分配率和提升率的波动大。如果发现这种情况，你应该查看 GC Timeline，找到垃圾收集活动的峰值。

默认时，GC Timeline 显示整个时间线上所有垃圾收集的停顿。

6.3.2 JConsole

JConsole 是一个 JMX（Java Management Extensions）兼容的 GUI 工具，可以连接运行中的 Java5 或者更高版本的 JVM。用 Java5 JVM 启动 Java 应用时，命令行只有添加-Dcom.sun.management.jmxremote 后，JConsole 才能连接，而 Java6 或更高版本的 JVM 不需要添加此属性。

JConsole 可以以三种方式连接正在运行的 JVM。

- Local：使用 JConsole 连接一个正在本地系统运行的 JVM，并且执行程序和运行 JConsole 的需要是同一个用户。JConsole 使用文件系统的授权通过 RMI 连接器连接到平台的 MBean 服务器上。这种从本地连接的监控能力只有 Sun 的 JDK 具有。
- Remote：使用 URL 通过 RMI 连接器连接到一个 JMX 代理，service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi。hostName 填入主机名称，portNum 为 JMX 代理启动时指定的端口。JConsole 为建立连接，需要在环境变量中设置 mx.remote.credentials 来指定用户名和密码从而进行授权。
- Advanced：使用一个特殊的 URL 连接 JMX 代理。一般情况是使用自己定制的连接器而不是 RMI 提供的连接器，或者是一个使用 JDK1.4 的实现了 JMX 和 JMX Rmote 的应用。

JConsole 工具在 JDK/bin 目录下，启动 JConsole 后，将自动搜索本机运行的 jvm 进程，不需要 jps 命令来查询指定。双击其中一个 jvm 进程即可开始监控，也可使用“远程进程”来连接远程服务器，如图 6-17 所示。

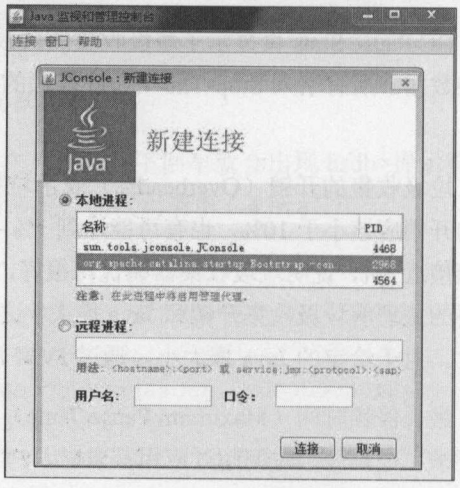


图 6-17 登录 JConsole

进入 JConsole 主界面，有“概述”、“内存”、“线程”、“类”、“VM 摘要”和“Mbean”等六个选项卡（TAB 页），如图 6-18 所示。

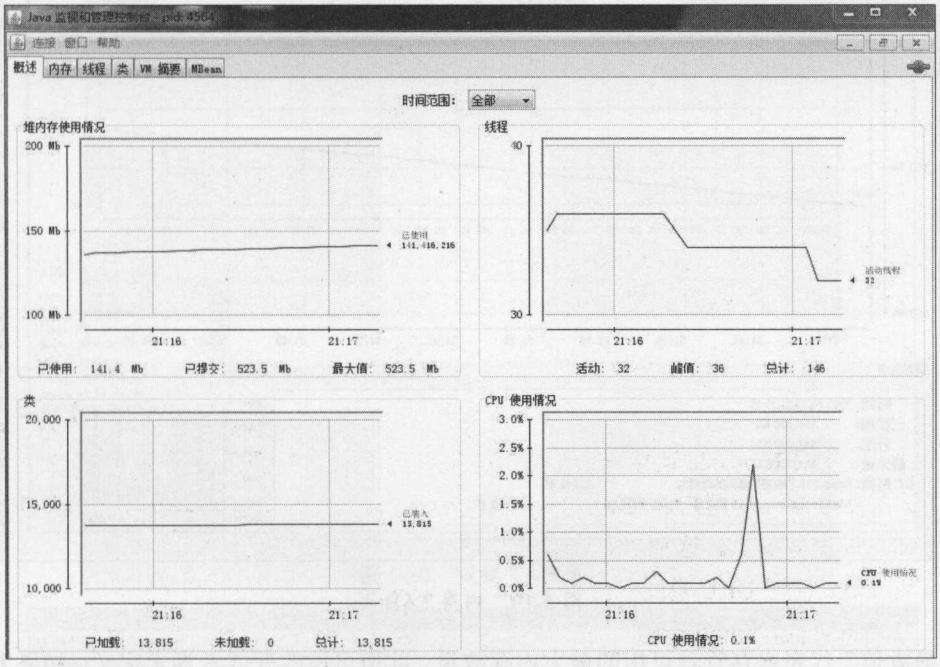


图 6-18　JConsole 主界面

概要选项卡显示了关于线程使用、内存消耗和 class 加载的一些关键监视信息，以及 JVM 和操作系统的信息。其中 Uptime 代表 JVM 已运行时长，Total compile time 代表花费在即时编译（JIT compilation）中的时间，Process CPU time 代表 JVM 花费的总 CPU 时间。

内存选项卡相当于 jstat 命令，用于监视收集器管理的虚拟机内存（Java 堆和永久代）变化趋势，还可在详细信息栏观察全部 GC 执行的时间及次数，如图 6-19 所示。

详细信息中“已使用”代表当前使用的内存总量。使用的内存总量是指所有的对象占用的内存，包括可达和不可达的对象。

“分配”代表 JVM 可使用的内存量。committed 内存数量可能随时间变化而变化。JAVA 虚拟机可能将某些内存释放，还给操作系统，committed 内存可能比启动时初始分配的内存量要少。committed 内存总是大于或者等于 used 内存。

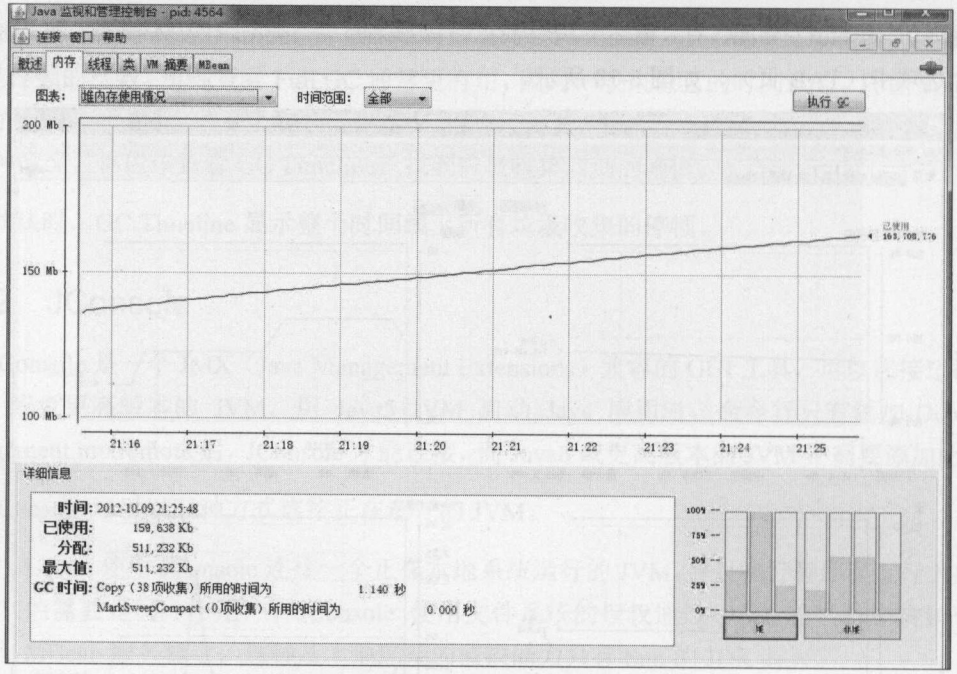


图 6-19　内存 TAB 页

“最大值”代表内存管理可用的最大内存数量。此值可能改变或者为未定义。如果 JVM 试图增加使用内存（used memory）超出了 committed 内存，那么即使使用内存小于或者等于最大内存（比如系统虚拟内存较低），内存分配仍可能失败。

线程选项卡，如图 6-20 所示，其中 Live threads 代表当前活动的 daemon 线程数量加 non-daemon 线程数量。Peak 代表自 JVM 启动后，活动线程峰值。Daemon threads 代表当前活动的 Daemon 线程数量。Total started 代表自 JVM 启动后，启动的线程总量（包括 daemon、non-daemon 和终止了的）。

最后一个常用选项卡是 VM 选项卡，可以清楚地了解、显示指定的 JVM 参数及堆信息，如图 6-21 所示。

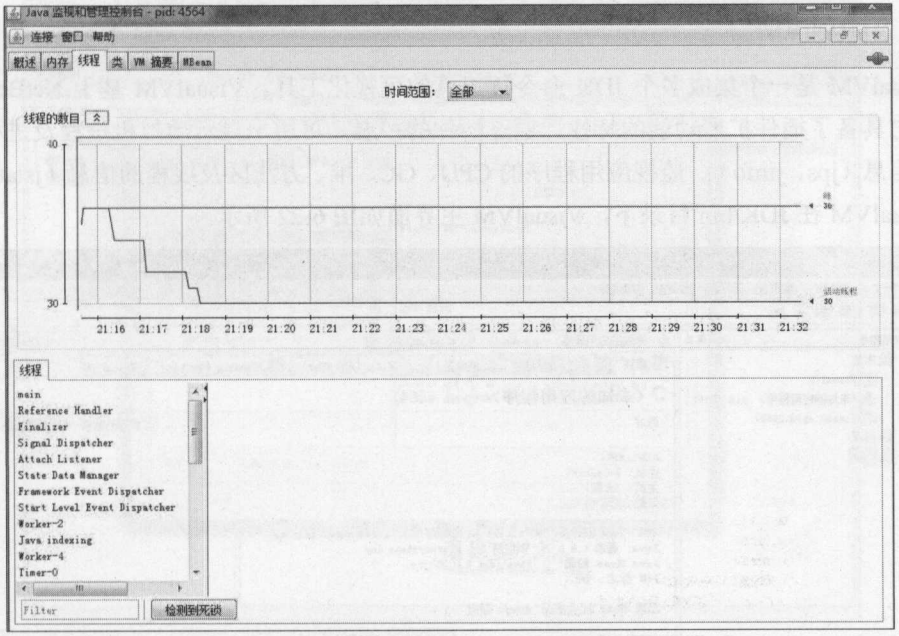


图 6-20　线程 TAB 页

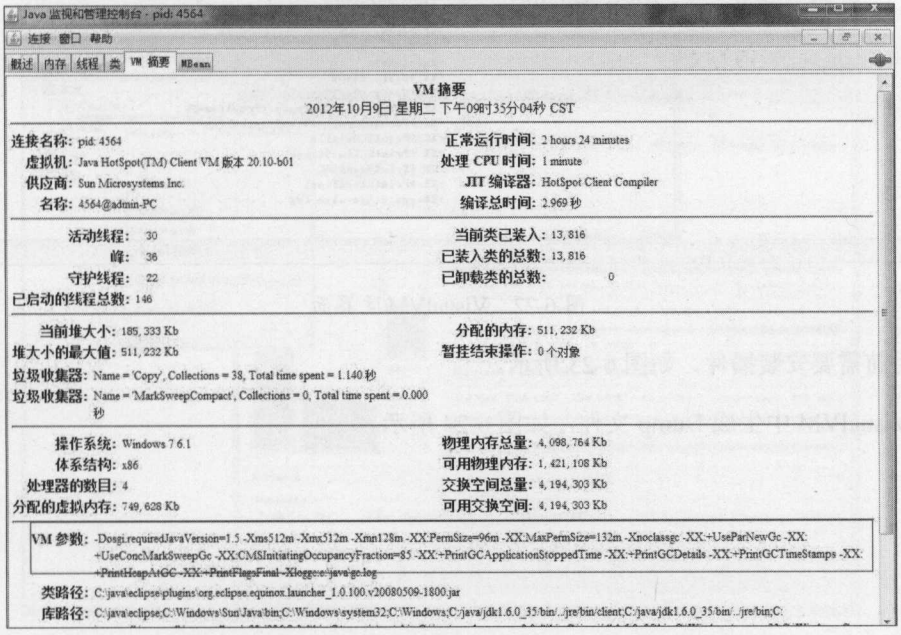


图 6-21　VM 信息 TAB 页

6.3.3 VisualVM

VisualVM 是一个集成多个 JDK 命令行工具的可视化工具。VisualVM 基于 NetBeans 平台开发，它具备了插件扩展功能的特性，通过插件的扩展，可用于显示虚拟机进程及进程的配置和环境信息（jps，jinfo），监视应用程序的 CPU、GC、堆、方法区及线程的信息（jstat、jstack）等。VisualVM 在 JDK/bin 目录下，VisualVM 主界面如图 6-22 所示。

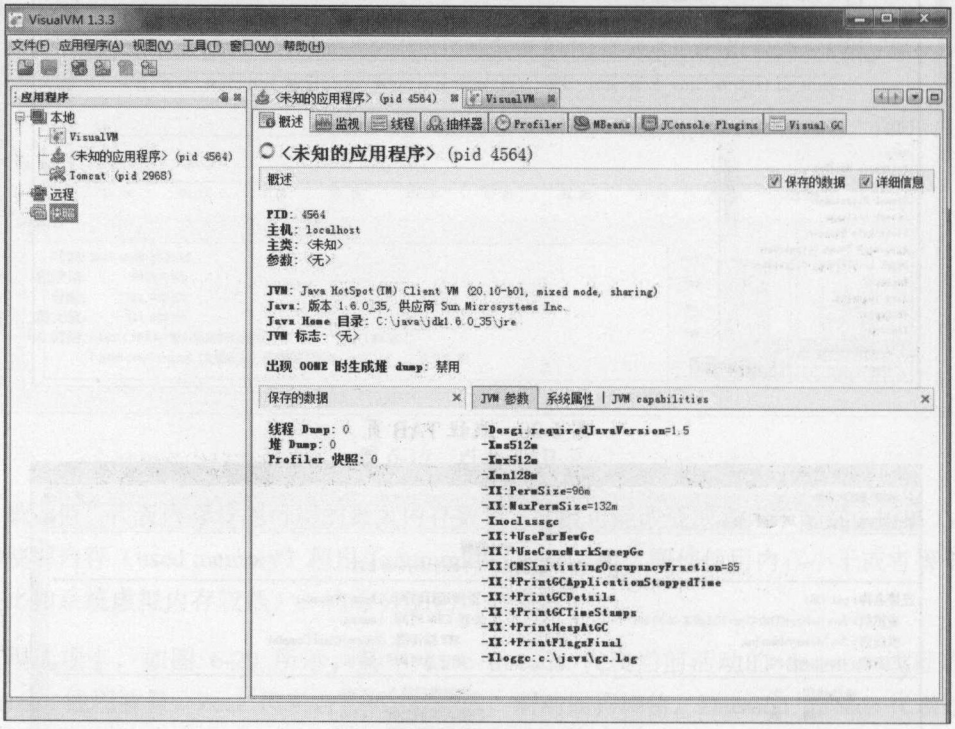


图 6-22　VisualVM 主界面

使用前需要安装插件，如图 6-23 所示。

在 VisualVM 中生成 Dump 文件，如图 6-24 所示。

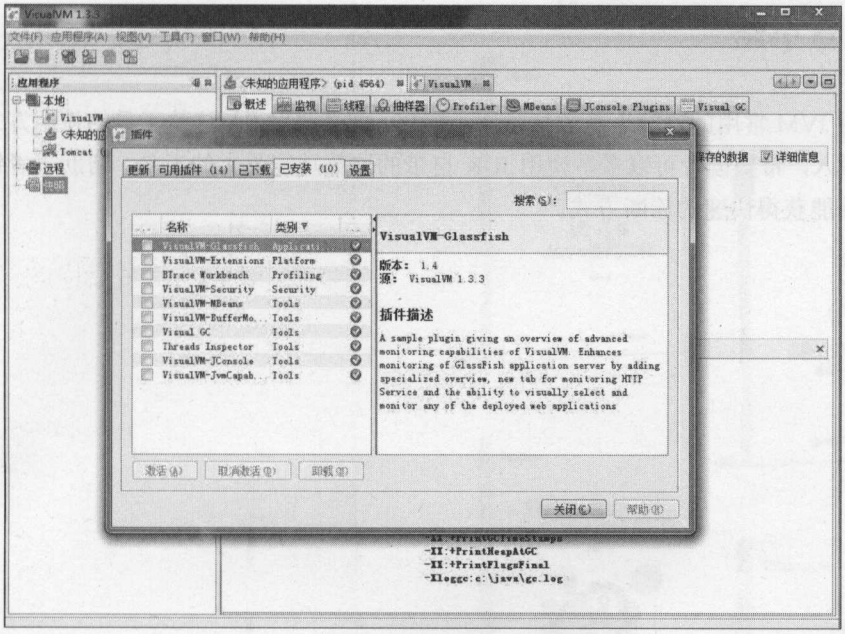


图 6-23　VisualVM 安装

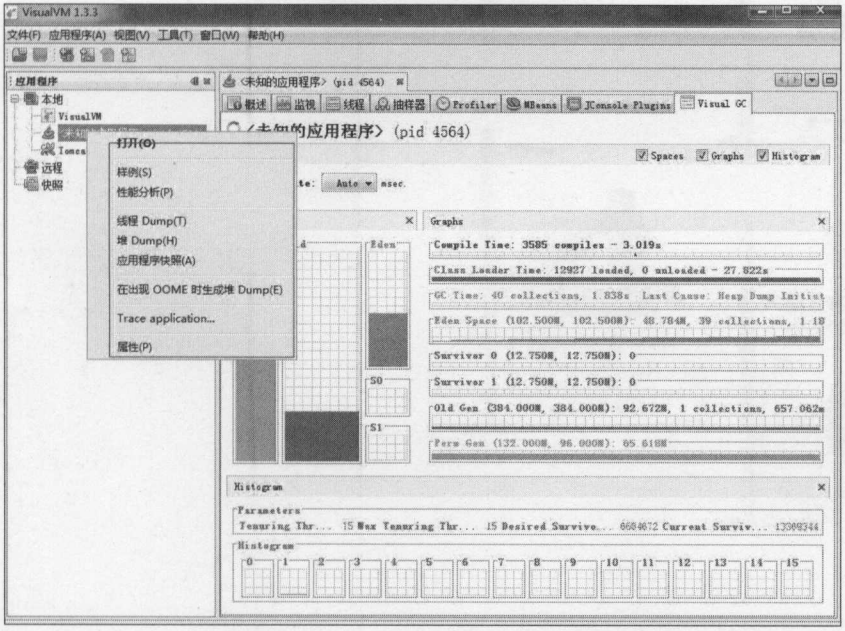
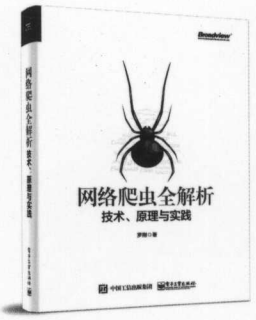


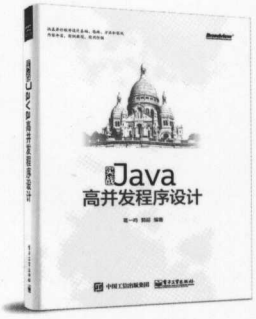
图 6-24　VisualVM 生成 Dump 文件

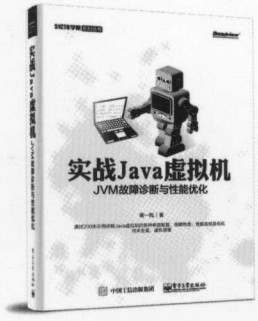
6.4 本章小结

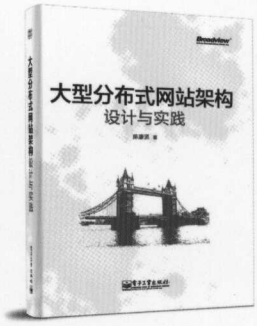
本章对 JVM 常用的诊断工具进行了逐一解释，尤其是对 SA 这款工具的使用方式介绍得最为全面、深入，希望读者可以多多使用 JDK 自带的这款功能强大的工具，帮助读者针对自己的应用程序性能获得快速的诊断方式。

好书分享









欢迎反馈意见或投稿
邮箱：dongying@phei.com.cn
电话：010-88254047
微信号：yingzidd

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

深入理解JVM & G1 GC

总的来说，本书对Java GC机制的分析深入浅出，是对大数据Java内存回收的优秀实践。读完茅塞顿开、受益匪浅。很多技术细节应用之后，对产品性能有明显提升。在此感谢周明耀的分享，希望他能够写出更多优秀的书籍。

华为南京研究所大数据产品部维护经理　吴骏

每年都要面试很多学生，我感觉中国的大学不太注重实际项目开发能力的培养，较为教条，这也是我的系列丛书首先从Java技术开始的原因，它更加接地气。本书主要为学习Java语言的学生、初级程序员提供JVM和GC的使用和优化建议及经验，力求做到知识的综合传播，而不是仅仅针对Java虚拟机调优进行讲解。本书具体包括以下几方面：JVM基础知识、GC基础知识、G1 GC的深入介绍、G1 GC调优建议、JDK自带工具使用介绍等。

周明耀

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：董　英
封面设计：李　玲

上架建议：计算机>Java

ISBN 978-7-121-31468-1
9 787121 314681 >
定价：69.00元